Προχωρημένα Θέματα Βάσεων Δεδομένων

Εξαμηνιαία Εργασία
στο Spark

[Github Repository link : εδώ]

Ομάδα 34

Δημήτριος-Δαυίδ Γεροκωνσταντής (ΑΜ : 03119209)
Αθανάσιος Τσουκλείδης-Καρυδάκης (ΑΜ : 03119009)

Χειμερινό 2023

Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή ΗΜΜΥ

Περιεχόμενα

Εισαγωγή

Σκοπός της παρούσας εξαμηνιαίας εργασίας είναι η αξιοποίηση εργαλείων διαχείρισης μεγάλου όγκου δεδομένων σε ένα κατανεμημένο περιβάλλον. Συγκεκριμένα, με χρήση του προγραμματιστικού μοντέλου του Spark, του resource manager YARN και του κατανεμημένου file system HDFS σκοπός είναι να κάνουμε - μέσω κατάλληλων queries - processing ενός μεγάλου όγκου dataset που θα αφορά τα εγκλήματα που διαπράχθηκαν στην κομητεία του Los Angeles από το 2010 έως σήμερα (αρχές Δεκεμβρίου 2023). Προς αυτή την κατεύθυνση, αρχικά θα δημιουργήσουμε ένα cluster από 2 nodes στο οποίο θα εγκατασταθεί το Spark με τον κατάλληλο resource manager όπως επίσης και το Hadoop DFS (το οποίο θα χρησιμοποιηθεί ως πηγή των δεδομένων μας). Στη συνέχεια, θα γίνει χρήση αυτού του συστήματος για κατανεμημένη εκτέλεση queries με χρήση διάφορων Spark APIs (RDDs, Dataframes, SQL). Ο κώδικας θα γραφεί με χρήση python (στο περιβάλλον pyspark). Στη συνέχεια, απαντώνται τα ερωτήματα της εξαμηνιαίας εργασίας κατά σειρά.

I. Ερώτημα 1^o : Διαμόρφωση του Συστήματος

Αρχικά ακολουθούμε πιστά τον οδηγό που δίνεται για την διαμόρφωση του cluster με χρήση πόρων του okeanos όπως επίσης και για την μετέπειτα εκγατάσταση των Spark, Yarn, HDFS. Η διαδικασία δεν αναλύεται εδώ, καθώς περιγράφεται στον οδηγό (**εδώ**). Επιπλέον, σηκώνουμε τις 3 απαιτούμενες Web εφαρμογές :

HDFS : http://83.212.81.77:9870 Για περιήγηση στο distributed filesystem και έλεγχο της κατάστασης και λειτουργίας του. []

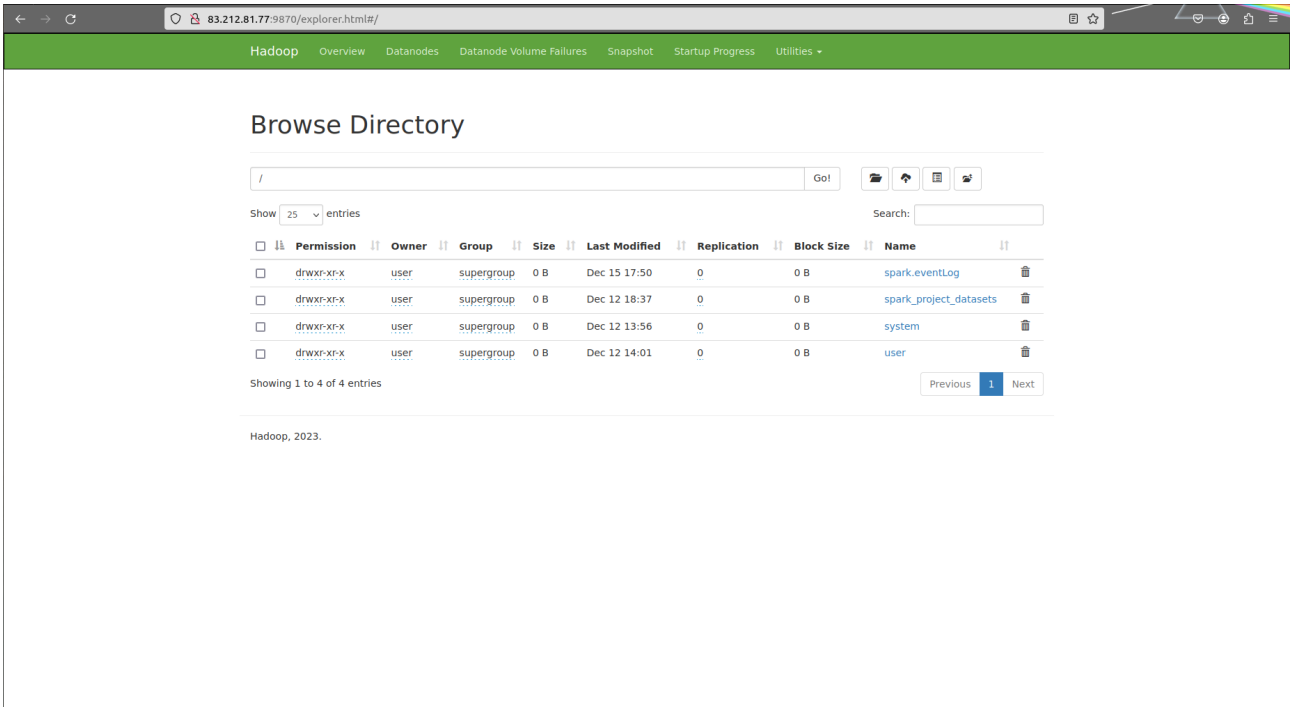

Figure 1: Η Web εφαρμογή του HDFS

YARN : http://83.212.81.77:8088 Η Web εφαρμογή του YARN.

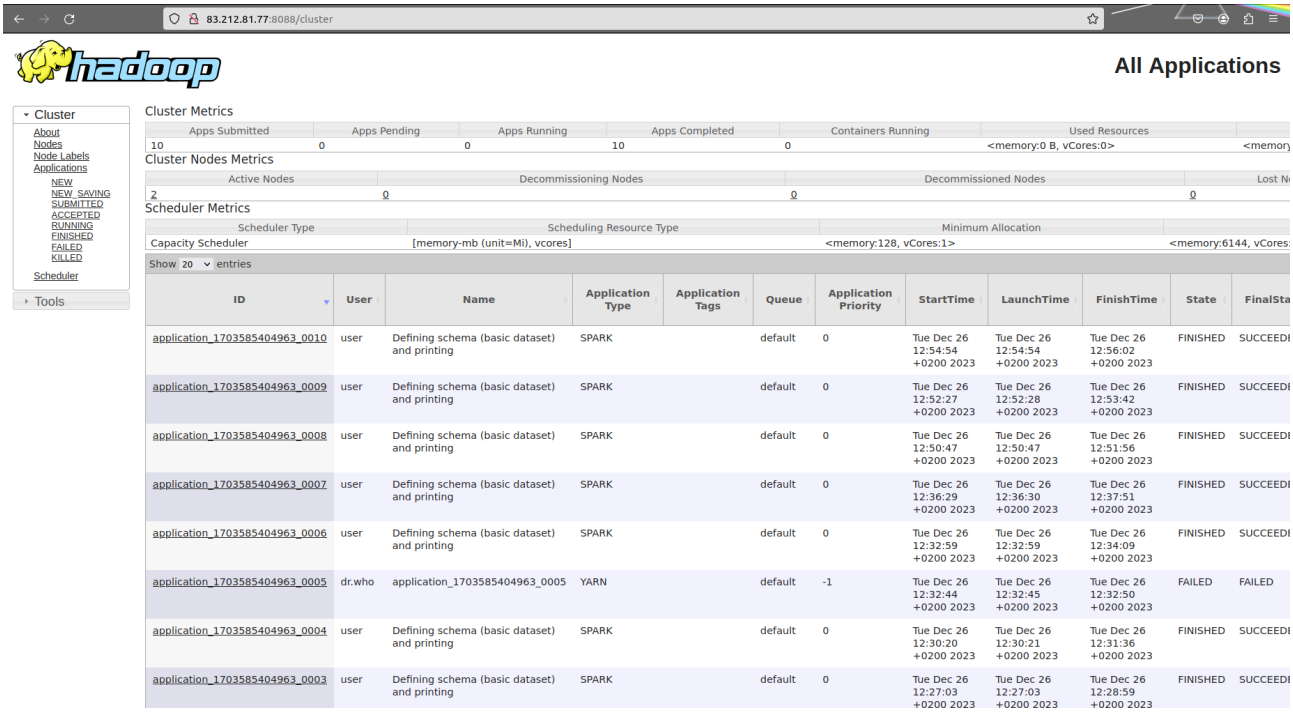

Figure 2: Η Web εφαρμογή του YARN

History Server : http://83.212.81.77:18080 Η Web εφαρμογή του History Server για έλεγχο των εργασιών που εκτελέστηκαν και ανάκτηση πληροφοριών για αυτές. Σημειώνουμε

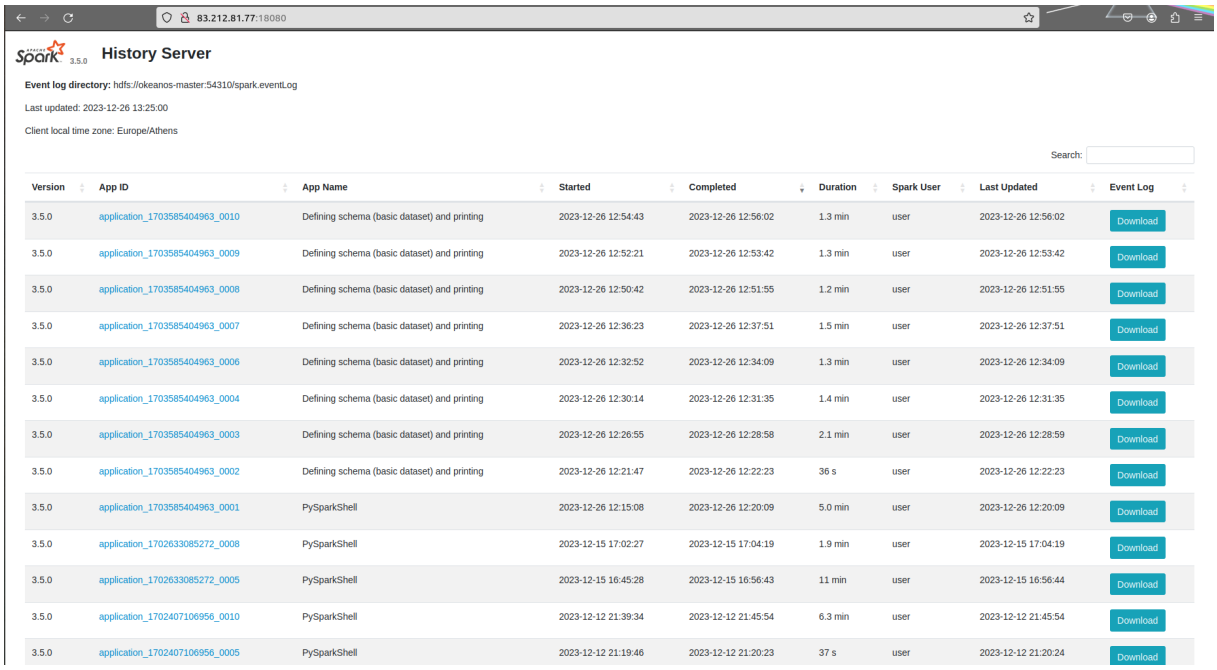

Figure 3: Η Web εφαρμογή του History Server

ότι εκτός των παραπάνω εγκαταστάθηκε στο cluster μας το jupyter notebook έτσι ώστε να χρησιμοποιούμε το περιβάλλον του pyspark στο notebook και όχι στο (κάπως άβολο) shell. Πλέον με την εντολή pyspark, προωθούμαστε στο jupyter notebook και για να γίνει αυτό, στο bashrc προστέθηκαν τα παρακάτω :

```
1    export PYSPARK_DRIVER_PYTHON='jupyter'
2    export PYSPARK_DRIVER_PYTHON_OPTS='notebook --no-browser --port=8889'
```

Επιπλέον, για να μπορέσουμε μέσω του προσωπικού μας υπολογιστή να αποκτήσουμε πρόσβαση στην κίνηση της πόρτας 8889 του remote okeanos node στην οποία τρέχει το jupyter, χρειάστηκε η ασφαλής μεταφορά αυτής της κίνησης στην αντίστοιχη πόρτα του προσωπικού μας υπολογιστή μέσω ssh tunneling. Αυτό επιτυγχάνεται με σύνδεση στο VM μέσω της εντολής :

```
ssh -i ~/.ssh/id_rsa -L 8889:localhost:8889 user@snf-39515.ok-kno.
    grnetcloud.net
```

ώστε με χρήση του ιδιωτικού μας κλειδιού (στο directory που δίνεται ως όρισμα δίπλα από το flag -i), να γίνει η σύνδεση στο αντίστοιχο VM με ssh tunneling. Για να ανοίξουμε το jupyter ορίζοντας κατάλληλο αριθμό από workers και μεταφέροντας στους nodes τα κατάλληλα πακέτα (που χρειάζεται να κάνουμε import αργότερα [1]), χρησιμοποιούμε την εντολή :

```
pyspark --conf spark.executor.instances=4 --py-files ./my_files/geopy2.zip
    ,./my_files/geographiclib.zip
```

στην οποία τα .zip αρχεία που φαίνονται περιλαμβάνουν τα αντίστοιχα απαραίτητα πακέτα (το πακέτο geographiclib απαιτείται από το geopy). Στη συνέχεια, πρέπει να φορτώσουμε στο HDFS τα απαραίτητα datasets. Τα φορτώνουμε αρχικά με wget (και το αντίστοιχο link στην διαδικτυακή τοποθεσία όπου βρίσκονται) τοπικά στον master node και στη συνέχεια τα φορτώνουμε στο HDFS με την εντολή :

```
hadoop fs -put <local source directory of the file> <hdfs destination
    directory>
```

Για την διαχείριση του συστήματος αρχείων του HDFS, χρήσιμες επίσης είναι οι εντολές

```
#for creating a new folder in hdfs
hadoop fs -mkdir <folder to be created>

#for deleting a file or folder from hdfs
hadoop fs -rm <file or folder to be removed>
```

Στη συνέχεια, αφού το HDFS περιλαμβάνει τα απαραίτητα datasets, οφείλουμε να κατανείμουμε το φορτίο μεταξύ των δύο nodes (διότι διαφορετικά, όλα τα αρχεία θα φορτωθούν στον master καθώς ενώ το hdfs είναι κατανεμειμένο, τα αρχεία μας είναι σχετικά μικρά και ο balancer δεν θα δουλέψει αυτόματα αλλά απαιτείται δική μας παρέμβαση). Συγκεκριμένα, τρέχουμε τον balancer με μικρότερο threshold (το οποίο είναι η επιτρεπόμενη ποσοστιαία απόκλιση στο φόρτο δεδομένων που αναλαμβάνει κάθε κόμβος). Απαιτούμε αυτό να μην είναι 10% (καθώς λόγω των μικρών αρχείων που διαθέτουμε, αυτό δεν θα κατανείμει τίποτα μεταξύ των κόμβων) αλλά 1% (δηλαδή το μικρότερο δυνατό). Χρησιμοποιούμε την εντολή :

```
hdfs balancer -threshold 1
```

Μετά από τα παραπάνω, το hdfs περιλαμβάνει τα αρχεία της εικόνας 4 ενώ με χρήση της εντολής:

[1] αναφερόμαστε στο πακέτο geopy που χρειάζεται για τον υπολογισμό απόστασης μεταξύ σημείων ορισμένων μέσω γεωγραφικών συντεταγμένων στο query 4

```
hdfs fsck /spark_project_datasets -files -blocks -locations
```

μπορούμε να δούμε στην εικόνα 5 την κατανομή των blocks (block size = 128MB) των datasets στους δύο nodes μας. Για παράδειγμα το αρχείο basic_dataset_2010to2019b.csv έχει μοιραστεί με 3 blocks να βρίσκονται στον master (192.168.0.2) και με 2 blocks να βρίσκονται στον worker (192.168.0.3)

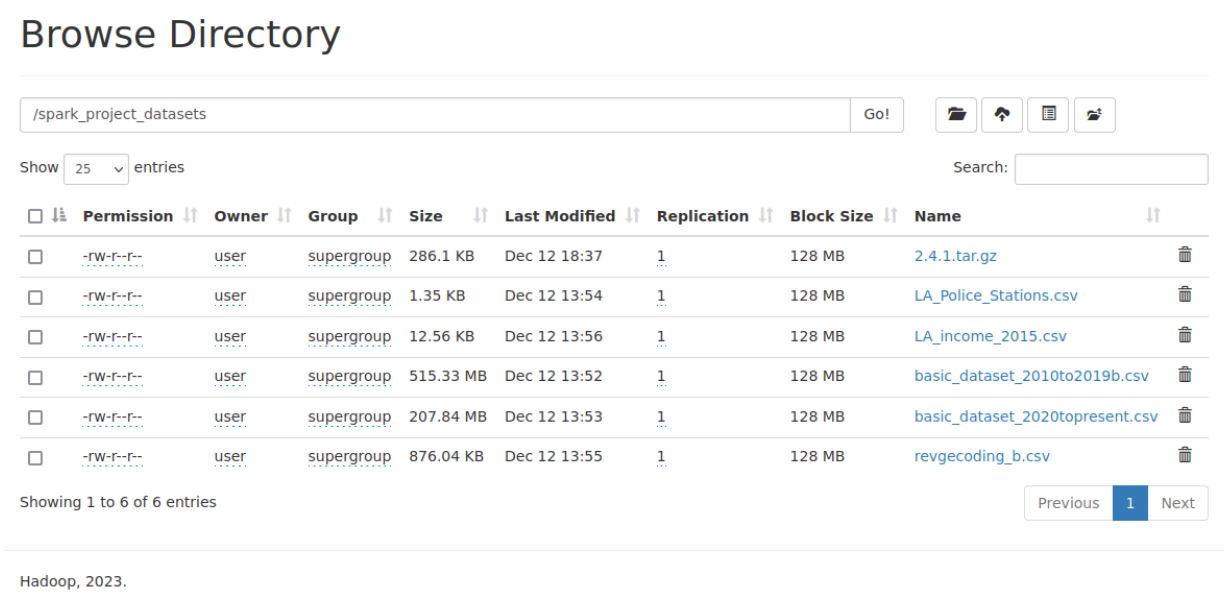

Figure 4: Τα περιεχόμενα του HDFS

```
user@okeanos-master:~$ hdfs fsck /spark_project_datasets -files -blocks -locations
Connecting to namenode via http://okeanos-master:9870/fsck?ugi=user&files=1&blocks=1&locations=1&path=%2Fspark_project_datasets
FSCK started by user (auth:SIMPLE) from /192.168.0.2 for path /spark_project_datasets at Tue Dec 26 13:59:25 EET 2023

/spark_project_datasets <dir>
/spark_project_datasets/2.4.1.tar.gz 292965 bytes, replicated: replication=1, 1 block(s):  OK
0. BP-389632768-192.168.0.2-1702381095414:blk_1073741879_1055 len=292965 Live_repl=1  [DatanodeInfoWithStorage[192.168.0.2:9866,DS-3a69a402-0d49-4203-a801-a1916d1f303c,DISK]]

/spark_project_datasets/LA_Police_Stations.csv 1387 bytes, replicated: replication=1, 1 block(s):  OK
0. BP-389632768-192.168.0.2-1702381095414:blk_1073741832_1008 len=1387 Live_repl=1  [DatanodeInfoWithStorage[192.168.0.2:9866,DS-3a69a402-0d49-4203-a801-a1916d1f303c,DISK]]

/spark_project_datasets/LA_income_2015.csv 12859 bytes, replicated: replication=1, 1 block(s):  OK
0. BP-389632768-192.168.0.2-1702381095414:blk_1073741834_1010 len=12859 Live_repl=1  [DatanodeInfoWithStorage[192.168.0.3:9866,DS-a56b03a2-980b-49cf-9b9f-68efe15e538f,DISK]]

/spark_project_datasets/basic_dataset_2010to2019b.csv 540364277 bytes, replicated: replication=1, 5 block(s):  OK
0. BP-389632768-192.168.0.2-1702381095414:blk_1073741825_1001 len=134217728 Live_repl=1  [DatanodeInfoWithStorage[192.168.0.3:9866,DS-a56b03a2-980b-49cf-9b9f-68efe15e538f,DISK]]
1. BP-389632768-192.168.0.2-1702381095414:blk_1073741826_1002 len=134217728 Live_repl=1  [DatanodeInfoWithStorage[192.168.0.3:9866,DS-a56b03a2-980b-49cf-9b9f-68efe15e538f,DISK]]
2. BP-389632768-192.168.0.2-1702381095414:blk_1073741827_1003 len=134217728 Live_repl=1  [DatanodeInfoWithStorage[192.168.0.2:9866,DS-3a69a402-0d49-4203-a801-a1916d1f303c,DISK]]
3. BP-389632768-192.168.0.2-1702381095414:blk_1073741828_1004 len=134217728 Live_repl=1  [DatanodeInfoWithStorage[192.168.0.2:9866,DS-3a69a402-0d49-4203-a801-a1916d1f303c,DISK]]
4. BP-389632768-192.168.0.2-1702381095414:blk_1073741829_1005 len=3493365 Live_repl=1  [DatanodeInfoWithStorage[192.168.0.2:9866,DS-3a69a402-0d49-4203-a801-a1916d1f303c,DISK]]

/spark_project_datasets/basic_dataset_2020topresent.csv 217936389 bytes, replicated: replication=1, 2 block(s):  OK
0. BP-389632768-192.168.0.2-1702381095414:blk_1073741830_1006 len=134217728 Live_repl=1  [DatanodeInfoWithStorage[192.168.0.3:9866,DS-a56b03a2-980b-49cf-9b9f-68efe15e538f,DISK]]
1. BP-389632768-192.168.0.2-1702381095414:blk_1073741831_1007 len=83718661 Live_repl=1  [DatanodeInfoWithStorage[192.168.0.3:9866,DS-a56b03a2-980b-49cf-9b9f-68efe15e538f,DISK]]

/spark_project_datasets/revgecoding_b.csv 897062 bytes, replicated: replication=1, 1 block(s):  OK
0. BP-389632768-192.168.0.2-1702381095414:blk_1073741833_1009 len=897062 Live_repl=1  [DatanodeInfoWithStorage[192.168.0.2:9866,DS-3a69a402-0d49-4203-a801-a1916d1f303c,DISK]]
```

Figure 5: Κατανομή των blocks των datasets στους 2 nodes

II. Ερώτημα 2^o : Δημιουργία βασικού Dataframe

Στο ερώτημα αυτό δημιουργούμε ένα dataframe με το συνολικό dataset που προκύπτει από συνένωση των 2 datasets (2010-2019 και 2020-present). Αφού δημιουργήσουμε το spark session και κάνουμε import τα κατάλληλα data types και συναρτήσεις (col), διαβάζουμε με read.csv τα δύο αρχεία από το hdfs χωρίς να έχουμε ορίσει προηγουμένως το schema του dataframe. Το σχήμα θα γίνει αυτόματα infer από το spark και στη συνέχεια για τα columns που μας ενδιαφέρουν, μέσω type casting (συνάρτηση cast()), αλλάζουμε τον τύπο τους όπως υποδεικνύεται. Με την union() ενώνουμε τα δύο datasets και τυπώνουμε το σχήμα και το πλήθος των γραμμών του συνολικού dataset. Αυτή η διαδικασία δημιουργίας του βασικού dataframe επαναλαμβάνεται και για όλα τα επόμενα queries. Ο κώδικας φαίνεται παρακάτω :

```
1  from pyspark.sql import SparkSession
2  from pyspark.sql.types import StructField, StructType, IntegerType,
       DoubleType, DateType
3  from pyspark.sql.functions import col
4
5  spark = SparkSession \
6      .builder \
7      .appName("Defining schema (basic dataset) and printing") \
8      .getOrCreate()
9
10 #reading the first part of the dataset and casting
11 basic_dataset_2010to2019_df = spark.read.csv("hdfs://okeanos-master:54310/
       spark_project_datasets/basic_dataset_2010to2019b.csv", header=True)
12 basic_dataset_2010to2019_df_casted = basic_dataset_2010to2019_df \
13 .withColumn("Date Rptd", col("Date Rptd").cast(DateType())) \
14 .withColumn("DATE OCC", col("DATE OCC").cast(DateType())) \
15 .withColumn("Vict Age", col("Vict Age").cast(IntegerType())) \
16 .withColumn("LAT", col("LAT").cast(DoubleType())) \
17 .withColumn("LON", col("LON").cast(DoubleType()))
18
19 #reading the second part of the dataset and casting
20 basic_dataset_2020topresent_df = spark.read.csv("hdfs://okeanos-master
       :54310/spark_project_datasets/basic_dataset_2020topresent.csv", header=
       True)
21 basic_dataset_2020topresent_df_casted = basic_dataset_2020topresent_df \
22 .withColumn("Date Rptd", col("Date Rptd").cast(DateType())) \
23 .withColumn("DATE OCC", col("DATE OCC").cast(DateType())) \
24 .withColumn("Vict Age", col("Vict Age").cast(IntegerType())) \
25 .withColumn("LAT", col("LAT").cast(DoubleType())) \
26 .withColumn("LON", col("LON").cast(DoubleType()))
27
28 #unify the 2 parts
29 full_dataset_df = basic_dataset_2010to2019_df_casted.union(
       basic_dataset_2020topresent_df_casted)
30
31 print('Schema :')
32 full_dataset_df.printSchema()
33
34 print('Number of Rows :')
35 print(full_dataset_df.count())
36
37 spark.stop()
```

Λαμβάνουμε τα εξής αποτελέσματα :

```
Schema :
root
 |-- DR_NO: string (nullable = true)
 |-- Date Rptd: date (nullable = true)
 |-- DATE OCC: date (nullable = true)
 |-- TIME OCC: string (nullable = true)
 |-- AREA : string (nullable = true)
 |-- AREA NAME: string (nullable = true)
 |-- Rpt Dist No: string (nullable = true)
 |-- Part 1-2: string (nullable = true)
 |-- Crm Cd: string (nullable = true)
 |-- Crm Cd Desc: string (nullable = true)
 |-- Mocodes: string (nullable = true)
 |-- Vict Age: integer (nullable = true)
 |-- Vict Sex: string (nullable = true)
 |-- Vict Descent: string (nullable = true)
 |-- Premis Cd: string (nullable = true)
 |-- Premis Desc: string (nullable = true)
 |-- Weapon Used Cd: string (nullable = true)
 |-- Weapon Desc: string (nullable = true)
 |-- Status: string (nullable = true)
 |-- Status Desc: string (nullable = true)
 |-- Crm Cd 1: string (nullable = true)
 |-- Crm Cd 2: string (nullable = true)
 |-- Crm Cd 3: string (nullable = true)
 |-- Crm Cd 4: string (nullable = true)
 |-- LOCATION: string (nullable = true)
 |-- Cross Street: string (nullable = true)
 |-- LAT: double (nullable = true)
 |-- LON: double (nullable = true)

Number of Rows :
2988445
```

Όπως φαίνεται, οι τύποι όλων των columns έχουν γίνει infer αυτόματα ως strings, εκτός αυτών για τα οποία ρητά ορίσαμε κάτι διαφορετικό. Το πλήθος των γραμμών συμφωνεί με αυτό που φαίνεται στις ιστοσελίδες από όπου λήφθηκαν τα datasets.

III. Ερώτημα 3^o: Query 1 σε DataFrame και SQL APIs

Υλοποιούμε αυτο το query σε DataFrame και SQL APIs και το εκτελούμε με 4 executors.

1. DataFrame API

Αρχικά, διαβάζουμε όπως στο ερώτημα 2, το βασικό μας dataset δημιουργώντας το αντίστοιχο DataFrame. Στη συνέχεια προχωρούμε στην υλοποίηση του query. Σκοπός μας είναι η εμφάνιση (και το ranking) ανά έτος των τριών μηνών με τα περισσότερα καταγεγραμμένα εγκλήματα. Για αυτό το σκοπό, θα χρησιμοποιηθεί window function, η οποία διευκολύνει το ranking μεταξύ των τριών αυτών μηνών. Αρχικά, αφού προσθέσουμε στο βασικό dataset τις στήλες year και month που εξάγονται από το πεδίο **DATE OCC** μέσω των συναρτήσεων **year()** και **month()**, ορίζουμε το specification του window function (δηλαδή ομαδοποίηση κατά year και ordering κατά το πλήθος των εγκλημάτων). Στη συνέχεια, στο κύριο μέρος του query, ομαδοποιούμε κατά year και month (έχοντας πλέον records για κάθε μήνα κάθε έτους), υπολογίζουμε με **count()** το πλήθος των εγκλημάτων σε κάθε μήνα κάθε έτους και εφαρμόζουμε την window function η οποία ομαδοποιεί κατά έτος και κάνει ranking (κατά φθίνουσα σειρα) των μηνών (κάθε έτους) βάσει του πλήθους εγκλημάτων που διαπράχθηκαν κατά τη διάρκειά τους. Επειδή θέλουμε τους top 3 μήνες, φιλτράρουμε εκείνα τα records που έχουν ranking 1,2 ή 3. Ο κώδικας που περιγράφηκε φαίνεται παρακάτω :

```
#add year and month columns for each dataset record
full_dataset_withYearandMonth = full_dataset \
.withColumn("year", year("DATE OCC")).withColumn("month", month("DATE OCC"
   ))

#defining the window function specifications
windowSpec = Window.partitionBy("year").orderBy(col("crime_total").desc())

#main body of the query
full_dataset_withYearandMonth_Window = full_dataset_withYearandMonth \
.groupBy(col("year"),col("month")).agg(count("*").alias("crime_total")).
   withColumn("#",rank().over(windowSpec)).where(col("#")<4).show(42)
```

2. SQL API

Με πολύ παρόμοιο τρόπο υλοποιείται το query 1 σε SQL API. Η διαφορά έγκειται στο γεγονός ότι πρέπει να δημιουργήσουμε SQL table από το βασικό dataframe ώστε να γράψουμε το SQL query, η λογική του οποίου είναι πανομοιότυπη με ό,τι αναφέρθηκε παραπάνω, αλλά σε σύνταξη SQL. Ο κώδικας φαίνεται παρακάτω :

```
#creating an SQL table from the dataframe of the main dataset
full_dataset_withYearandMonth_df.createOrReplaceTempView("full_dataset")

#the main query
id_query = "SELECT * FROM \
    (SELECT year, month, count(*) as crime_total, rank() OVER w as rank
       FROM full_dataset \
    GROUP BY year, month \
```

```
    WINDOW w AS (PARTITION BY year ORDER BY count(*) DESC)) \
    WHERE rank<4"

results = spark.sql(id_query)
results.show(42)
```

3. Αποτελέσματα Query 1

Η εκτέλεση των παραπάνω εξήγαγε τα αποτελέσματα που παρατίθενται στη συνέχεια και τα οποία έχουν την αναμενόμενη μορφή. Για παράδειγμα, μπορούμε να δούμε ότι το 2019, ο μήνας με τα περισσότερα εγκλήματα ήταν ο Ιούλιος ενώ ακολούθησε ο Αύγουστος και έπειτα ο Μάρτιος.

```
+----+-----+-----------+----+
|year|month|crime_total|rank|
+----+-----+-----------+----+
|2010|    1|      19515|   1|
|2010|    3|      18131|   2|
|2010|    7|      17856|   3|
|2011|    1|      18134|   1|
|2011|    7|      17283|   2|
|2011|   10|      17034|   3|
|2012|    1|      17943|   1|
|2012|    8|      17661|   2|
|2012|    5|      17502|   3|
|2013|    8|      17440|   1|
|2013|    1|      16820|   2|
|2013|    7|      16644|   3|
|2014|    7|      12196|   1|
|2014|   10|      12133|   2|
|2014|    8|      12028|   3|
|2015|   10|      19219|   1|
|2015|    8|      19011|   2|
|2015|    7|      18709|   3|
|2016|   10|      19659|   1|
|2016|    8|      19490|   2|
|2016|    7|      19448|   3|
|2017|   10|      20431|   1|
|2017|    7|      20192|   2|
|2017|    1|      19833|   3|
|2018|    5|      19972|   1|
|2018|    7|      19875|   2|
|2018|    8|      19761|   3|
|2019|    7|      19121|   1|
|2019|    8|      18979|   2|
|2019|    3|      18854|   3|
|2020|    1|      18496|   1|
|2020|    2|      17255|   2|
|2020|    5|      17204|   3|
|2021|   10|      26676|   1|
|2021|   12|      26317|   2|
|2021|   11|      25715|   3|
|2022|    5|      20418|   1|
|2022|   10|      20274|   2|
```

```
|2022|     6|       20201|   3|
|2023|     8|       19743|   1|
|2023|     7|       19697|   2|
|2023|     1|       19633|   3|
+----+------+------------+----+
```

4. Επιδόσεις - Συγκρίσεις

Στους παραπάνω κώδικες φροντίζουμε με χρήση της βιβλιοθήκης time, να προσθέσουμε κατάλληλους timers για τη μέτρηση του χρόνου εκτέλεσης. Αυτός ο χρόνος θα περιλαβάνει όλο το σώμα του κώδικα (μαζί με το διάβασμα των datasets και το show() στο τέλος), σύμβαση που διατηρούμε σε όλη μας την εργασία ώστε τα αποτελέσματα να είναι συγκρίσιμα. Για λόγους διακύμανσης του χρόνου εκτέλεσης μεταξύ διαφορετικών μετρήσεων (ειδικά όταν γίνονται σε εντελώς διαφορετικές στιγμές) που πηγάζουν από τον τρόπο λειτουργίας του συστήματος (π.χ. αλλαγή του φόρτου του συστήματος σε διαφορετικές χρονικές στιγμές), φροντίζουμε να κάνουμε επαρκείς μετρήσεις και σε διαφορετικές στιγμές. Συγκεκριμένα, παρακάτω φαίνονται 7 μετρήσεις που λήφθηκαν σε τυχαίες ώρες διαφορετικών ημερών. Αυτοί οι χρόνοι, για το **DataFrame API** είναι οι εξής:

Μέτρηση 1	39.87169170379639
Μέτρηση 2	37.33209252357483
Μέτρηση 3	36.779780864715576
Μέτρηση 4	31.913537740707397
Μέτρηση 5	34.13872265815735
Μέτρηση 6	35.075082540512085
Μέτρηση 7	32.53778791427612

Για να λάβουμε μία αντιπροσωπευτική τιμή χρόνου εκτέλεσης, θα υπολογίσουμε έναν μέσο όρο των παραπάνω μετρήσεων, αφαιρώντας την μέγιστη και την ελάχιστη τιμή (θεωρώντας τα outliers). Λαμβάνουμε την τιμή :

$$\frac{37.332 + 36.77978 + 34.1387 + 35.075 + 32.5377879}{5} = 35.1726933sec$$

Για το **SQL API** είναι οι παρακάτω:

Μέτρηση 1	33.14069890975952
Μέτρηση 2	26.451717138290405
Μέτρηση 3	21.280057907104492
Μέτρηση 4	33.26009440422058
Μέτρηση 5	35.2558376789093
Μέτρηση 6	35.40161657333374
Μέτρηση 7	32.18000555038452

Ομοίως, η αντιπροσωπευτική τιμή που λαμβάνουμε είναι η :

$$\frac{33.14 + 26.4517 + 33.26 + 35.2558 + 32.18}{5} = 32,057670736sec$$

Παρατηρούμε ότι ενώ οι μετρήσεις για τα δύο APIs δεν έχουν σημαντικές διαφορές, το SQL API φαίνεται να επιτυγχάνει λίγο καλύτερη επίδοση. Και τα δύο APIs αποτελούν ένα high level abstraction του RDD API (με το SQL API να είναι ακόμη πιο high level) και χρησιμοποιούν το ίδιο execution engine (SPARK execution engine) και τον ίδιο optimizer (Catalyst). Ως προς το optimization, ενώ χρησιμοποιείται ο ίδιος optimizer, στην περίπτωση των DataFrames μπορεί να επιτευχθεί fine grained control over optimizations καθώς ο προγραμματιστής ορίζει ρητά transformations (steps μετασχηματισμού των datasets) και μπορεί να κατευθύνει τον optimizer και τη δημιουργία του execution plan (π.χ. με χρήση της συνάρτησης cache()). Έτσι, εν δυνάμει, αν ο προγραμματιστής κάνει καλό χειρισμό των DataFrames μπορεί να υπάρξει ένα καλώς βελτιστοποιημένο execution plan. Ωστόσο, ο explicit ορισμός των βημάτων εκτέλεσης (transformations) μπορεί να περιορίσει τις δυνατότητες του built-in optimizer δημιουργώντας ένα λιγότερο αποδοτικό πλάνο εκτέλεσης. Σε αντίθεση με το DataFrame API, στο SQL API -μέσω της SQL- με declarative τρόπο ορίζουμε αυτό που θέλουμε να κάνουμε retrieve από τη βάση μας (και όχι επακριβώς μέσω explicitly defined αλυσίδας από transformations). Με αυτό τον τρόπο η αποδοτικότητα του παραγόμενου πλάνου εκτέλεσης επαφίεται εξ'ολοκλήρου στον Catalyst Optimizer. Αξίζει να σημειωθεί ωστόσο ότι με χρήση της explain(True), συγκρίναμε τα optimized logical plans των δύο APIs και δεν παρατηρήθηκαν ουσιώδεις διαφορές. Η τελευταία διαφορά στην οποία θα μπορούσαν να οφείλονται οι μικρές διαφορές στο χρόνο εκτέλεσης, αφορά το code generation process, κατά το οποίο το πλάνο εκτέλεσης μετατρέπεται σε bytecode που πρόκειται να εκτελεστεί από τους nodes. Ενώ στο SQL API παράγεται bytecode κατά το χρόνο εκτέλεσης (runtime) ώστε τα SQL queries να εκτελούνται αποδοτικά, το DataFrame API βασίζεται στο JVM για να εκτελέσει transformations, με συνέπεια να υπάρχουν πιθανές διαφοροποιήσεις στην επίδοση. Τελικά, η επιλογή μεταξύ των δύο APIs δεν βασίζεται τόσο στην διαφορά επίδοσης όσο σε ζητήματα που αφορούν την εξοικείωση του προγραμματιστή με καθένα από τα δύο, την ευκολία προγραμματισμού (και την ευκολία για debugging : σε SQL API τα συντακτικά errors γίνονται catch at runtime ενώ στα DataFrames εκ των προτέρων at compile time).

IV. Ερώτημα 4ο: Query 2 σε DataFrame/SQL και RDD APIs

Υλοποιούμε αυτο το query σε DataFrame/SQL και RDD APIs και το εκτελούμε με 4 executors. Στο github link που δίνεται, υπάρχει η υλοποίηση τόσο για DataFrame όσο και για SQL API, ωστόσο όπως διαπιστώθηκε, αυτά τα δύο δεν διαφέρουν ουσιαστικά οπότε παρακάτω παρουσιάζεται και επεξηγείται μόνο η DataFrame υλοποίηση.

1. DataFrame/SQL API

Διαβάζουμε ως γνωστόν, το βασικό μας dataset δημιουργώντας το αντίστοιχο DataFrame και προχωρούμε στην υλοποίηση του query. Δημιουργούμε μια User Defined Function η οποία λαμβάνει ως είσοδο το timestamp στο οποίο διαπράχθηκε το εκάστοτε έγκλημα και επιστρέφει την περίοδο της μέρας στην οποία ανήκει αυτό. Αυτή η function μετατρέπεται μέσω της **udf()** σε udf χρησιμοποιούμενη από DataFrames και χρησιμοποιείται ώστε να δημιουργηθεί για κάθε crime record ένα νέο πεδίο με την περίοδο της μέρας κατά τη διάρκειας της οποίας διαπράχθηκε. Φιλτράρουμε τα rows που αφορούν crimes που διαπράχθηκαν στο δρόμο, ομαδοποιούμε κατά περίοδο της ημέρας, υπολογίζουμε τον αριθμό των εγκλημάτων για κάθε περίοδο της ημέρας (με **count()**) και κατατάσσουμε τις περιόδους της μέρας κατά φθίνων πλήθος εγκλημάτων. Ο κώδικας φαίνεται παρακάτω :

```
#defining the User Defined Function that matches a specific time to its
    corresponding period of day
def get_period_of_day(time_occ):
    if int(time_occ)>=500 and int(time_occ)<=1159:
        return "05:00-11:59"
    elif int(time_occ)>=1200 and int(time_occ)<=1659:
        return "12:00-16:59"
    elif int(time_occ)>=1700 and int(time_occ)<=2059:
        return "17:00-20:59"
    elif (int(time_occ)>=0 and int(time_occ)<=459) or (int(time_occ)>=2100
        and int(time_occ)<=2359) :
        return "21:00-03:59"
    else :
        return "Null"

#register the UDF
get_period_of_day_udf = udf(get_period_of_day, StringType())

#add a new column with the corresponding period of day (about when the
    crime was commited)
full_dataset_df_withDayPeriod = full_dataset_df \
    .withColumn("period_of_day", get_period_of_day_udf(col("TIME OCC")))

#main body of the query
full_dataset_df_withDayPeriod.filter(col("Premis Desc")=="STREET").groupBy
    ("period_of_day").agg(count("*") \
.alias("crime_total")).orderBy(col("crime_total").desc()).select("
    period_of_day", "crime_total").show()
```

Ο κώδικας σε SQL API είναι εντελώς παρόμοιας λογικής, με το χρησιμοποιούμενο SQL query να έχει ως εξής :

```
id_query = """
SELECT period_of_day, count(*) AS crime_total
FROM full_dataset_withDayPeriod
WHERE `Premis Desc`='STREET'
GROUP BY period_of_day
ORDER BY crime_total DESC
"""
```

2. RDD API

Στην υλοποίηση με RDDs, η προσέγγιση είναι κάπως διαφορετική στο διάβασμα των csv αρχείων που περιλαμβάνουν τα βασικά datasets. Ενώ σε αυτές τις περιπτώσεις, για το διάβασμα csv αρχείων, κάνουμε split με βάση το "," (παίρνοντας έτσι ένα σύνολο από λίστες, μια για κάθε row με όλα τα πεδία του row), εδώ δυστυχώς κάποια πεδία (όπως το "Crm Cd Desc") παρατηρήσαμε ότι υπάρχει περίπτωση να περιέχουν "," ως μέρος του value τους χωρίς αυτό όμως να σημαίνει ότι εκεί πρέπει να γίνει split. Αυτό είχε ως συνέπεια (όταν κάναμε split) το 15ο πεδίο που χρειάζεται να ελέγξουμε αν είναι ίσο με "STREET", στις συγκεκριμένες προβληματικές rows, να μην βρίσκεται στην θέση 15 αλλά σε επόμενες από αυτήν. Για να μην οδηγηθούμε σε λανθασμένα αποτελέσματα, το διάβασμα των csv αρχείων έγινε με τη βοήθεια του έτοιμου csv reader που δίνεται από την βιβλιοθήκη **csv** την οποία κάναμε import. Για το κύριο μέρος του query, στόχος είναι να αντιστοιχίσουμε κάθε crime του dataset με την περίοδο της μέρας στην οποία διαπράχθηκε. Για αυτό το σκοπό, χρησιμοποιείται η map η οποία για κάθε είσοδο x του dataset που διαβάζει (δηλαδή για κάθε row) επιστρέφει μια τούπλα με πρώτο στοιχείο την αντίστοιχη περίοδο της μέρας και ως δέυτερο στοιχείο έναν άσο (1) που υποδηλώνει ότι συναντήθηκε αυτή η περίοδος της μέρας μια φορά. Για την αντιστοίχιση αυτή ελέγχεται το 3ο πεδίο (και συγκεκριμένα οι πρώτοι δύο χαρακτήρες του που αντιστοιχούν στην ώρα) και εντοπίζεται ποιας περιόδου ώρα αναπαριστά (με βάση τις λίστες που ορίστηκαν παραπάνω) και βέβαια ελέγχεται αν το 15ο πεδίο ("Premis Desc") είναι ίσο με "STREET" καθώς αυτά τα rows ας ενδιαφέρουν. Τυχόντα rows των οποίων η ώρα δε δίνεται, αντοιστοιχίζονται σε None και στη συνέχεια φιλτράρονται και απομακρύνονται από το RDD. Με τα παραπάνω, πλέον έχουμε key value pairs από περιόδους της μέρας (key) και άσους (1 ως value) και μπορούμε να ομαδοποιήσουμε ως προς key, αθροίζοντας τους άσους που αντιστοιχούν σε ένα key. Τέλος, πλέον καταλήγυμε με key value pairs που έχουν ως key την περίοδο της μέρας και value το άθροισμα των φορών όπου αυτές συναντήθηκαν στο dataset και αρκεί να ταξινομήσουμε ως προς αυτό το value (που είναι το x[1]) κατά φθίνουσα σειρά. Με **collect()** συλλέγουμε τα αποτελέσματά μας.

```
import csv
from io import StringIO
from pyspark.sql import SparkSession
import time

#function to parse the csv file (because of a problem explained in the
    report)
def parse_csv(row):
    csv_reader = csv.reader(StringIO(row))
    return next(csv_reader)

sc = SparkSession \
    .builder \
```

```
13      .appName("RDD query 2 execution") \
14      .getOrCreate() \
15      .sparkContext
16
17  start=time.time()
18
19  # reading the datasets
20  basic_dataset_2010to2019 = sc.textFile("hdfs://okeanos-master:54310/
        spark_project_datasets/basic_dataset_2010to2019b.csv") \
21                  .map(parse_csv)
22  basic_dataset_2020topresent= sc.textFile("hdfs://okeanos-master:54310/
        spark_project_datasets/basic_dataset_2020topresent.csv") \
23                  .map(parse_csv)
24
25  full_datasets = basic_dataset_2010to2019.union(basic_dataset_2020topresent
        )
26
27
28  #defining the different periods of day
29  morning_list=['05','06','07','08','09','10','11']
30  afternoon_list = ['12','13','14','15','16']
31  evening_list = ['17','18','19','20']
32  night_list = ['21','22','23','00','01','02','03','04']
33
34
35  #the main body of the query
36  Morning_RDD_counter = full_datasets\
37  .map(lambda x: ['morning', 1] if((x[3][:2] in morning_list) and (x[15]=='
        STREET')) \
38                              else ['afternoon', 1] if((x[3][:2] in
                                    afternoon_list) and (x[15]=='STREET'))  \
39                              else ['evening', 1] if((x[3][:2] in
                                    evening_list) and (x[15]=='STREET'))  \
40                              else ['night', 1] if((x[3][:2] in night_list)
                                    and (x[15]=='STREET')) \
41                              else None) \
42  .filter(lambda x: x != None)\
43  .reduceByKey(lambda x,y:x+y).sortBy(lambda x:x[1],ascending=False).collect
        ()
44
45  print(Morning_RDD_counter)
46
47  end=time.time()
48
49  print("Execution time: ",end-start)
50  sc.stop()
```

3. Αποτελέσματα Query 2

Η εκτέλεση των παραπάνω εξήγαγε τα αποτελέσματα που παρατίθενται στη συνέχεια και τα οποία έχουν την αναμενόμενη μορφή, τόσο με DataFrames όσο και με RDDs. Τα αποτελέσματα και με τις δύο μεθόδους είναι ίδια και συγκεκριμένα κατατάσσουν τις περιόδους της μέρας κατά φθίνων πλήθος εγλημάτων στο δρόμο ως εξής : Νύχτα, Βράδυ, Απόγευμα, Πρωί.

DataFrames/SQL Το αποτέλεσμα εκτέλεσης με DataFrames/SQL είναι το παρακάτω:

```
+-------------+-----------+
|period_of_day|crime_total|
+-------------+-----------+
|  21:00-03:59|     237137|
|  17:00-20:59|     186896|
|  12:00-16:59|     148077|
|  05:00-11:59|     123748|
+-------------+-----------+
```

RDDs Το αποτέλεσμα εκτέλεσης με RDDs είναι το παρακάτω:

```
[('night', 237137), ('evening', 186896), ('afternoon', 148077), ('morning
    ', 123748)]
```

4. Επιδόσεις - Συγκρίσεις

Ομοίως προσθέτουμε κατάλληλους timers για τη μέτρηση του χρόνου εκτέλεσης και κρατάμε αρκετές μετρήσεις για λόγους που εξηγήθηκαν. Συγκεκριμένα, παρακάτω φαίνονται 7 μετρήσεις που λήφθηκαν σε τυχαίες ώρες διαφορετικών ημερών. Αυτοί οι χρόνοι, για το **DataFrame API** είναι οι εξής:

Μέτρηση 1	20.73853850364685
Μέτρηση 2	30.438623428344727
Μέτρηση 3	20.99086833000183
Μέτρηση 4	23.71483325958252
Μέτρηση 5	29.999660968780518
Μέτρηση 6	23.42507815361023
Μέτρηση 7	30.42247486114502

Για να λάβουμε μία αντιπροσωπευτική τιμή χρόνου εκτέλεσης, θα υπολογίσουμε έναν μέσο όρο των παραπάνω μετρήσεων, αφαιρώντας την μέγιστη και την ελάχιστη τιμή (θεωρώντας τα outliers). Λαμβάνουμε την τιμή :

$$\frac{20.99 + 23.71483 + 29.9996609 + 23.425 + 30.42247}{5} = 25.710583115sec$$

Για το **RDD API** είναι οι παρακάτω:

Μέτρηση 1	37.88229966163635
Μέτρηση 2	31.473424673080444
Μέτρηση 3	36.40968704223633
Μέτρηση 4	31.965908527374268
Μέτρηση 5	37.349095582962036
Μέτρηση 6	31.989267826080322
Μέτρηση 7	34.91439867019653

Ομοίως, η αντιπροσωπευτική τιμή που λαμβάνουμε είναι η :

$$\frac{36.409687 + 31.9659 + 37.349095 + 31.9892678 + 34.914398}{5} = 34.52567153sec$$

Παρατηρούμε ότι η υλοποίηση με RDDs επιτυγχάνει χειρότερες επιδόσεις από αυτήν με DataFrames/ SQL. Τα RDDs αποτελούν έναν πολύ low level τρόπο για να κάνει κάποιος μετασχηματισμούς και υπολογισμούς επί των datasets. Ο προγραμματιστής ορίζει ακριβώς το "πώς" θα γίνει κάτι, ορίζοντας πλήρως την πορεία εκτέλεσης. Κατά αυτόν τον τρόπο, τα RDDs **δεν γίνονται optimized** (δεν περιλαμβάνουν κάποιον built in optimizer, όπως τον Catalyst) και είναι ευθύνη του προγραμματιστή να ορίσει τους μετασχηματισμούς με τέτοια σειρά ώστε το πρόγραμμα να είναι αποδοτικό (fine grained control, περισσότερο από τα DataFrames). Ως εκ τούτου είναι εύκολο μια υλοποίηση τελικά να αποδειχθεί μη αποδοτική. Επιπλέον, τα RDDs δεν επιτυγχάνουν καλές επιδόσεις σε non-JVM languages όπως η Python (που χρησιμοποιούμε), ενώ γενικά προτιμώνται όταν δουλεύουμε με unstructured data λόγω του τρόπου με τον οποίο αντιμετωπίζουν το schema των datasets. Για τους παραπάνω λόγους, οι επιδόσεις των RDDs είναι σαφώς χειρότερες από αυτές των DataFrames.

V. Ερώτημα 5^o: Query 3 σε DataFrame/SQL API

Υλοποιούμε αυτο το query σε DataFrame/SQL API χρησιμοποιώντας 2,3 και 4 executors. Αρχικά παρατείθεται και επεξηγείται ο κώδικας με DataFrames (παρόμοιας λογικής με SQL).

1. DataFrame/SQL API

Θα χρειαστούμε μια User Defined Function που αντιστοιχίζει τις συντομογραφίες των descents στην πλήρη ονομασία τους. Αφού οριστεί αυτή, διαβάζουμε τα datasets μας με κατάλληλο τρόπο όπως συνήθως. Αρχικά κάνουμε ένα reprocessing των δεδομένων μας. Αφήνουμε εκτός, τα records με null descent και από το reverse gecoding dataset κρατάμε για κάθε record μόνο το Zip Code χωρίς το postal suffix (που χρησιμοποιείται για περαιτέρω διαχωρισμό των υποπεριοχών μιας περιοχής). Στη συνέχεια, κάνουμε join το βασικό dataset με αυτό του reverse gecoding ώστε κάθε crime να συσχετιστεί με το Zip Code της περιοχής στην οποία διαπράχθηκε. Με αυτόν τον τρόπο επίσης πετυχαίνουμε η δημιουργηθείσα στήλη ZIPcode να περιέχει από το σύνολο των υπαρχόντων ZIP codes μόνο τα relevants, δηλαδή αυτά στα οποία πράγματι διαπράχθηκε κάποιο καταγεγραμμένο έγκλημα. Έπειτα, το νέο αυτό DataFrame γίνεται join με το DataFrame των median incomes ώστε κάθε έγκλημα να συσχετιστει όχι μόνο με το ZIP code της περιοχής όπου διαπράχθηκε αλλά και με το median income αυτής της περιοχής. Έτσι, από αυτό το DataFrame μπορούμε για κάθε crime να κρατήσουμε την καταγωγή του θύματος, το ZIP code της περιοχής όπου διαπράχθηκε και το median income αυτής. Από αυτό το DataFrame βρίσκουμε τα ZIP codes των τριών περιοχών με υψηλότερο και των τριών περιοχών με χαμηλότερο median income. Πλέον, έχουμε όλη την απαραίτητη πληροφορία για την υλοποίηση του κύριου μέρους του query μας. Επιλέγουμε να εμφανίσουμε 3 διαφορετικούς πίνακες. Αυτόν με το πλήθος θυμάτων ανά καταγωγή στις 3 "πλουσιότερες" περιοχές, έναν δεύτερο πίνακα με το πλήθος θυμάτων ανά καταγωγή στις 3 "φτωχότερες" περιοχές και έναν αθροιστικό τελευταίο πίνακα. Για τους πρώτους δύο πίνακες χρησιμοποιούμε το DataFrame που συσχέτιζε κάθε έγκλημα με την καταγωγή του θύματος και με το Zip code της περιοχής όπου διαπράχθηκε και το DataFrame με τα ZIP codes των 3 πλουσιότερων (αντίστοιχα φτωχότερων περιοχών). Κάνουμε join αυτών των δύο DataFrames, ομαδοποιούμε ως προς την καταγωγή και αθροίζουμε το πλήθος θυμάτων ανά καταγωγή ταξινομώντας τα αποτελέσματα όπως ζητείται. Ο τελικός πίνακας προκύπτει από union των δύο υποπινάκων, ομαδοποίηση ως προς την καταγωγή και άθροιση των πληθών θυμάτων ανά καταγωγή. Με παρόμοιο τρόπο υλοποιείται το query σε SQL, όπως φαίνεται στο github repository. Επιλέχθηκ για λόγους συντομίας να μην παρατεθεί εδώ.

```
1  from pyspark.sql import SparkSession
2  from pyspark.sql.types import StructField, StructType, IntegerType,
       DoubleType, DateType, StringType
3  from pyspark.sql.functions import col, to_timestamp, count, udf, split,
       monotonically_increasing_id,regexp_replace,min,sum,year
4  import time
5
6  spark = SparkSession \
7      .builder \
8      .appName("Defining schema (basic dataset) and printing") \
9      .getOrCreate()
10
11 #Function that matches descent abbreviations to the full descent names
```

```
12 def get_full_victim_descent(victim_descent):
13     if victim_descent=="A":
14         return "Other Asian"
15     elif victim_descent=="B":
16         return "Black"
17     elif victim_descent=="C":
18         return "Chinese"
19     elif victim_descent=="D":
20         return "Cambodian"
21     elif victim_descent=="F":
22         return "Filipino"
23     elif victim_descent=="G":
24         return "Guamanian"
25     elif victim_descent=="H":
26         return "Hispanic/Latin/Mexican"
27     elif victim_descent=="I":
28         return "American Indian/Alaskan Native"
29     elif victim_descent=="J":
30         return "Japanese"
31     elif victim_descent=="K":
32         return "Korean"
33     elif victim_descent=="L":
34         return "Laotian"
35     elif victim_descent=="O":
36         return "Other"
37     elif victim_descent=="P":
38         return "Pacific Islander"
39     elif victim_descent=="S":
40         return "Samoan"
41     elif victim_descent=="U":
42         return "Hawaiian"
43     elif victim_descent=="V":
44         return "Vietnamese"
45     elif victim_descent=="W":
46         return "White"
47     elif victim_descent=="X":
48         return "Unknown"
49     elif victim_descent=="Z":
50         return "Asian Indian"
51
52 get_full_victim_descent = udf(get_full_victim_descent, StringType())
53
54
55 start = time.time()
56
57 #reading the main datasets
58 basic_dataset_2010to2019_df = spark.read.csv("hdfs://okeanos-master:54310/
      spark_project_datasets/basic_dataset_2010to2019b.csv", header=True)
59 basic_dataset_2010to2019_df_casted = basic_dataset_2010to2019_df \
60 .withColumn("Date Rptd", to_timestamp("Date Rptd", "MM/dd/yyyy hh:mm:ss a"
      ).cast(DateType())) \
61 .withColumn("DATE OCC", to_timestamp("DATE OCC", "MM/dd/yyyy hh:mm:ss a").
      cast(DateType())) \
62 .withColumn("Vict Age", col("Vict Age").cast(IntegerType())) \
63 .withColumn("LAT", col("LAT").cast(DoubleType())) \
64 .withColumn("LON", col("LON").cast(DoubleType()))
```

```
65
66
67 basic_dataset_2020topresent_df = spark.read.csv("hdfs://okeanos-master
       :54310/spark_project_datasets/basic_dataset_2020topresent.csv", header=
       True)
68 basic_dataset_2020topresent_df_casted = basic_dataset_2020topresent_df \
69 .withColumn("Date Rptd", to_timestamp("Date Rptd", "MM/dd/yyyy hh:mm:ss a"
       ).cast(DateType())) \
70 .withColumn("DATE OCC", to_timestamp("DATE OCC", "MM/dd/yyyy hh:mm:ss a").
       cast(DateType())) \
71 .withColumn("Vict Age", col("Vict Age").cast(IntegerType())) \
72 .withColumn("LAT", col("LAT").cast(DoubleType())) \
73 .withColumn("LON", col("LON").cast(DoubleType()))
74
75
76 full_dataset_df = basic_dataset_2010to2019_df_casted.union(
       basic_dataset_2020topresent_df_casted)\
77 .withColumn("year",year("DATE OCC"))
78
79 #keep records with not null descent and only relevant to year 2015
80 full_dataset_noNullDescent_df=full_dataset_df.filter(col("Vict Descent").
       isNotNull()).filter(col("year")==2015)
81
82 #reading the other datasets
83 median_household_income_df = spark.read.csv("hdfs://okeanos-master:54310/
       spark_project_datasets/LA_income_2015.csv", header=True)\
84 .withColumn("Estimated median income",regexp_replace(col("Estimated Median
        Income"), "[^\d]", "").cast(IntegerType()))
85
86 reverse_geocoding_df = spark.read.csv("hdfs://okeanos-master:54310/
       spark_project_datasets/revgecoding_b.csv", header=True)
87
88 #keep only the Zip code value for the records that contain postal code
       suffix too (i.e. if we have 90731-7232 we only want to keep the 90731)
89 reverse_geocoding_keepFirstZip_df=reverse_geocoding_df.withColumn("
       ZIP_code",split(col("ZIPcode"),"-")[0]).filter(col("ZIP_code").
       isNotNull())
90
91
92 # purpose : keep only ZIP codes relevant to the basic dataset (in which a
       crime has been commited)
93 full_dataset_noNullDescent_withZip_df = full_dataset_noNullDescent_df.join
       (reverse_geocoding_keepFirstZip_df, \
94 (full_dataset_noNullDescent_df["LAT"]==reverse_geocoding_keepFirstZip_df["
       LAT"]) & \
95 (full_dataset_noNullDescent_df["LON"]==reverse_geocoding_keepFirstZip_df["
       LON"]),"leftouter") \
96 .select("Vict Descent","ZIPcode")
97 full_dataset_noNullDescent_withZip_df.persist()
98
99 #After this, for each crime we have the Victim Descent, the Zip Code and
       the median income for that zip code
100 full_dataset_noNullDescent_withZip_andIncome_df =
       full_dataset_noNullDescent_withZip_df.join(median_household_income_df,\
101 full_dataset_noNullDescent_withZip_df['ZIPcode']==
       median_household_income_df['Zip Code']) \
```

```
.select("Vict Descent", "ZIPcode","Estimated median income")

#keep Zip codes with the corresponding median incomes
income_perRelevant_zip_code_df =
    full_dataset_noNullDescent_withZip_andIncome_df.select("ZIPcode","
    Estimated median income")\
.groupBy("ZIPcode").agg(min("Estimated median income").alias("Estimated
    median income"))
income_perRelevant_zip_code_df.persist()

#keep Zip codes of areas with the lowest and highest median income
lowest_income_zip_codes3_df = income_perRelevant_zip_code_df.orderBy("
    Estimated median income").select("ZIPcode").limit(3)
highest_income_zip_codes3_df = income_perRelevant_zip_code_df.orderBy(col(
    "Estimated median income").desc()).select("ZIPcode").limit(3)

#main body of the query
print("For Highest Income")
Victim_Descent_Highest_Income = full_dataset_noNullDescent_withZip_df.join
    (highest_income_zip_codes3_df,"ZIPcode").groupBy("Vict Descent").agg(
    count("*").alias("#"))\
.orderBy(col("#").desc()).withColumn("Victim Descent",
    get_full_victim_descent(col("Vict Descent"))).select("Victim Descent","
    #")
Victim_Descent_Highest_Income.show()

print("For Lowest Income")
Victim_Descent_Lowest_Income = full_dataset_noNullDescent_withZip_df.join(
    lowest_income_zip_codes3_df,"ZIPcode").groupBy("Vict Descent").agg(
    count("*").alias("#"))\
.orderBy(col("#").desc()).withColumn("Victim Descent",
    get_full_victim_descent(col("Vict Descent"))).select("Victim Descent","
    #")
Victim_Descent_Lowest_Income.show()

print("Total Results")
Victim_Descent_Highest_Income.union(Victim_Descent_Lowest_Income).groupBy(
    "Victim Descent").agg(sum("#").alias("#")).orderBy(col("#").desc()).
    show()

end=time.time()

print("Execution Time:",end-start)

spark.stop()
```

2. Αποτελέσματα Query 3

Η εκτέλεση των παραπάνω εξήγαγε τα αποτελέσματα που παρατίθενται στη συνέχεια και τα οποία έχουν την αναμενόμενη μορφή.

```
For Highest Income
+--------------------+---+
|      Victim Descent|  #|
+--------------------+---+
```

```
|               White|338|
|               Other|106|
|Hispanic/Latin/Me...| 52|
|             Unknown| 27|
|               Black| 16|
|         Other Asian| 16|
+--------------------+---+


For Lowest Income
+--------------------+----+
|      Victim Descent|   #|
+--------------------+----+
|Hispanic/Latin/Me...|1531|
|               Black|1093|
|               White| 703|
|               Other| 393|
|         Other Asian| 103|
|             Unknown|  64|
|              Korean|   9|
|            Japanese|   3|
|American Indian/A...|   3|
|             Chinese|   2|
|            Filipino|   1|
+--------------------+----+


Total Results
+--------------------+----+
|      Victim Descent|   #|
+--------------------+----+
|Hispanic/Latin/Me...|1583|
|               Black|1109|
|               White|1041|
|               Other| 499|
|         Other Asian| 119|
|             Unknown|  91|
|              Korean|   9|
|American Indian/A...|   3|
|            Japanese|   3|
|             Chinese|   2|
|            Filipino|   1|
+--------------------+----+
```

3. Επιδόσεις - Συγκρίσεις

Ομοίως προσθέτουμε κατάλληλους timers για τη μέτρηση του χρόνου εκτέλεσης και κρατάμε αρκετές μετρήσεις για λόγους που εξηγήθηκαν. Συγκεκριμένα, παρακάτω φαίνονται 7 μετρήσεις για κάθε διαφορετικό πλήθος από executors οι οποίες λήφθηκαν σε τυχαίες ώρες διαφορετικών ημερών.

2 executors Αυτοί οι χρόνοι, για 2 executors είναι οι εξής:

Μέτρηση 1	47.29906368255615
Μέτρηση 2	35.532697916030884
Μέτρηση 3	39.81871938705444
Μέτρηση 4	36.26170229911804
Μέτρηση 5	38.66596341133118
Μέτρηση 6	35.70856261253357
Μέτρηση 7	33.76226329803467

Για να λάβουμε μία αντιπροσωπευτική τιμή χρόνου εκτέλεσης, θα υπολογίσουμε έναν μέσο όρο των παραπάνω μετρήσεων, αφαιρώντας την μέγιστη και την ελάχιστη τιμή (θεωρώντας τα outliers). Λαμβάνουμε την τιμή :

$$\frac{35.53269 + 39.818719 + 36.2617 + 38.66596 + 35.7085626}{5} = 37,197529125sec$$

Για μια από αυτές τις μετρήσεις παρατίθεται ενδεικτικά ο καταμερισμός των Tasks τους executors στην εικόνα 6.

Executors

Show 20 entries

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)
driver	okeanos-master:34307	Active	0	0.0 B / 434.4 MiB	0.0 B	0	0	0	0	0	1.2 min (0.0 ms)
1	okeanos-master:38589	Active	0	0.0 B / 434.4 MiB	0.0 B	1	0	0	67	67	24 s (0.4 s)
2	okeanos-worker:39431	Active	0	0.0 B / 434.4 MiB	0.0 B	1	0	0	27	27	35 s (1 s)

Figure 6: Καταμερισμός Tasks σε executors για 2 executors

3 executors Για 3 executors, οι χρόνοι είναι οι παρακάτω:

Μέτρηση 1	46.97204089164734
Μέτρηση 2	40.99207949638367
Μέτρηση 3	31.48922634124756
Μέτρηση 4	50.30695390701294
Μέτρηση 5	36.91513849263519
Μέτρηση 6	39.640440464019775
Μέτρηση 7	42.77695155143738

Ομοίως, η αντιπροσωπευτική τιμή που λαμβάνουμε είναι η :

$$\frac{46.972 + 40.992079 + 36.915138 + 39.64 + 42.776951}{5} = 41,459330179sec$$

Για μια από αυτές τις μετρήσεις παρατίθεται ενδεικτικά ο καταμερισμός των Tasks τους executors στην εικόνα 7.

Executors

Show 20 entries

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)
driver	okeanos-master:45843	Active	0	0.0 B / 434.4 MiB	0.0 B	0	0	0	0	0	1.4 min (0.0 ms)
1	okeanos-worker:34115	Active	0	0.0 B / 434.4 MiB	0.0 B	1	0	0	21	21	30 s (1 s)
2	okeanos-master:37945	Active	0	0.0 B / 434.4 MiB	0.0 B	1	0	0	43	43	19 s (0.4 s)
3	okeanos-worker:36885	Active	0	0.0 B / 434.4 MiB	0.0 B	1	0	0	16	16	29 s (1 s)

Figure 7: Καταμερισμός Tasks σε executors για 3 executors

4 executors Για 4 executors, οι χρόνοι είναι οι παρακάτω:

Μέτρηση 1	47.89033770561218
Μέτρηση 2	47.22955298423767
Μέτρηση 3	40.825082302093506
Μέτρηση 4	38.383941888809204
Μέτρηση 5	44.50140142440796
Μέτρηση 6	48.06897521018982
Μέτρηση 7	40.38305330276489

Ομοίως, η αντιπροσωπευτική τιμή που λαμβάνουμε είναι η :

$$\frac{47.89 + 47.22955 + 40.825082 + 44.5014 + 40.383}{5} = 44,165885544sec$$

Για μια από αυτές τις μετρήσεις παρατίθεται ενδεικτικά ο καταμερισμός των Tasks τους executors στην εικόνα 8.

Executors

Show 20 entries

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)
driver	okeanos-master:45293	Active	0	0.0 B / 434.4 MiB	0.0 B	0	0	0	0	0	1.5 min (0.0 ms)
1	okeanos-worker:37333	Active	0	0.0 B / 434.4 MiB	0.0 B	1	0	0	15	15	23 s (0.7 s)
2	okeanos-master:34929	Active	0	0.0 B / 434.4 MiB	0.0 B	1	0	0	34	34	16 s (0.4 s)
3	okeanos-worker:43397	Active	0	0.0 B / 434.4 MiB	0.0 B	1	0	0	22	22	32 s (1 s)
4	okeanos-master:37243	Active	0	0.0 B / 434.4 MiB	0.0 B	1	0	0	43	43	19 s (0.4 s)

Figure 8: Καταμερισμός Tasks σε executors για 4 executors

Από τις παραπάνω αντιπροσωπευτικές τιμές, παρατηρούμε ότι με αύξηση του αριθμού των χρησιμοποιούμενων executors αυξάνεται (!) ο χρόνος εκτέλεσης του προγράμματος. Εδώ, θα πρέπει να σκεφτούμε το trade off που εμφανίζεται όταν χρησιμοποιούμε πολλαπλούς executors. Από τη μία η αύξηση των executors εν δυνάμει μπορεί να βελτιώσει την επίδοση καθώς κατανέμει τον υπολογιστικό φόρτο μεταξύ τους, από την άλλη οι πολλαπλοί executors θα πρέπει να διαμοιραστούν τους κοινούς πόρους του συστήματος, μειώνοντας τελικά τους διαθέσιμους πόρους (π.χ. μνήμη, CPU time) ανά executor. Επιπλέον προστίθεται στο συνολικό κόστος, αυτό της διαχείρισης των πολλαπλών executors, της μεταξύ τους επικοινωνίας και του πιθανού contention (conflicts) στους διαμοιραζόμενους πόρους. Στην περίπτωση που η εφαμρογή μας περιλαμβάνει μεγάλο βαθμό παραλληλισμού και υψηλό task granularity (δηλαδή tasks που μπορούν να σπάσουν σε πολλά ανεξάρτητα παραλληλοποιήσιμα subtasks), η προσθήκη πολλαπλών executors μπορεί να βελτιώσει την επίδοση. Από την άλλη αν αυτά τα χαρακτηριστικά δεν είναι έντονα στην εφαρμογή μας, τελικά το κόστος των overheads που αναφέρθηκαν επικρατεί. Ένας επιπλέον κρίσιμος παράγοντας είναι ο τρόπος με τον οποίο τα δεδομένα μας είναι κατανεμημένα μεταξύ των διαφορετικών κόμβων στους οποίους τρέχουν οι executors. Για παράδειγμα, αν δεν είχαμε φροντίσει τον κατάλληλο καταμερισμό των δεδομένων, όλα τα δεδομένα θα βρίσκονταν σε έναν κόμβο, όλοι οι executors θα έτρεχαν σε αυτόν και τελικά θα γινόταν underutilized το σύστημά μας. Στην περίπτωσή μας κάτι τέτοιο δεν συμβαίνει καθώς όπως περιεγράφηκε στην αρχή, τα δεδομένα έχουν καταμεριστεί μεταξύ των workers. Αλλά και πάλι, τίποτα δεν μας εγγυάται ότι ο αυτόματος καταμερισμός που έγινε από τον balancer τελικά εξυπηρετεί την αποδοτική εκτέλεση των queries καθώς δεν μπορούμε ακριβώς να ξέρουμε πώς αυτά εκτελούνται από το σύστημα και πώς κατανέμεται η εκτέλεσή τους μεταξύ των nodes. Επιπλέον, ενώ τα "group by" και τα aggregations είναι inherently parallelizable, συνολικά η εφαρμογή μας ενδέχεται να μην έχει υψηλό task granularity και να μην σπάει αποδοτικά σε μικρά tasks. Ακόμα και τα δημιουργούμενα tasks που ανατίθενται στους executors ενδέχεται μεταξύ τους να μην είναι ανεξάρτητα, δημιουργώντας έτσι ένα σημαντικό σειριακό μέρος στην εκτέλεση. Και στην περίπτωσή μας ενώ μπορεί κάποια tasks να είναι παραλληλοποιήσιμα (όπως τα aggrgations), συνολικά το πρόγραμμά μας αποτελείται από επιμέρους βήματα κάθε ένα από τα οποία χρησιμοποιεί αποτελέσματα προηγούμενων βημάτων (διαδοχικά transfomations, κάθε ένα απο τα οποία εφαρμόζεται σε ένα προηγουμένως υπολογισμένο DataFrame). Αυτό οδηγεί σε ένα σημαντικό σειριακό μέρος στην εκτέλεση. Τέλος, μια συνιστώσα που θα μπορούσε να εξεταστεί ως προς την επίδρασή της στην επίδοση είναι το πλήθος των cores ανά executor (παράμετρος spark.executor.cores). Αν η εφαρμογή μας είχε σημαντικό παραλληλισμό, η χρήση της default τιμής 1, όπου σε κάθε executor ανατίθεται ένας υπολογιστικός πυρήνας, ενδεχομένως θα οδηγούσε σε υποχρησιμοποίηση (underutilization) των πόρων μας και τελικά μη αποδοτική εκμετάλλευση του παραλληλισμού. Δοκιμάσαμε να αυξήσουμε την παράμετρο στην τιμή 2, χωρίς αυτό να έχει κάποια επίδραση στον χρόνο εκτέλεσης. Αυτό οφείλεται στον τελικώς χαμηλό παραλληλισμό της εφαρμογής μας και στα σημαντικό σειριακό της μέρος, όπως εξηγήθηκε προηγουμένως. Στην εικόνα 9 μπορούμε να δούμε ότι οι executors δεν προστίθενται όλοι μαζί αλλά κάπως σειριακά, ενδεχομένως λόγω σειριακών τμημάτων της εφαμρογής μας.

Σημείωση Αξιοσημείωτη στις παραπάνω μετρήσεις είναι η έντονη διακύμανση των μετρήσεων χρόνου εκτέλεσης για 3 executors. Αυτό οφείλεται πιθανότατα στην ανισοκατανομή των executors στους nodes που δημιουργείται σε αυτήν την περίπτωση, καθώς έχουμε 3 executors και 2 nodes. Όπως φαίνεται και στην εικόνα 7, ενδεικτικά στην εκτέλεση που απεικονίζεται, η

κατανομή ήταν τέτοια ώστε 2 executors να τρέχουν στον worker και 1 στον master.

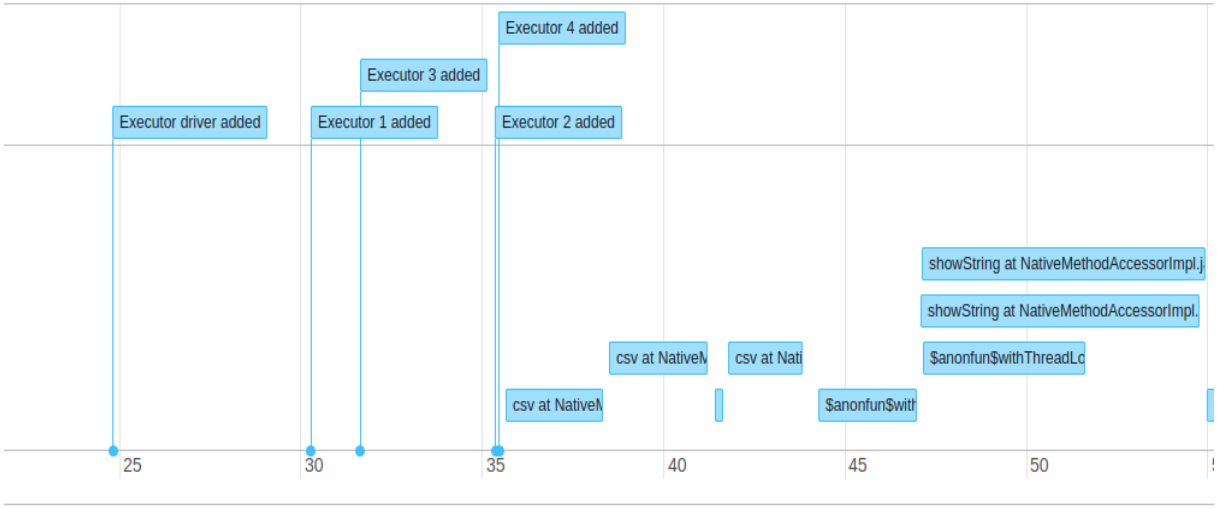

Figure 9: Event Timeline από τον History Server για 4 executors

VI. Ερώτημα 6^o: Query 4 σε DataFrame/SQL API

Υλοποιούμε αυτο το query σε DataFrame/SQL API χρησιμοποιώντας (ενδεικτικά) 4 executors. Αρχικά παρατείθεται και επεξηγείται ο κώδικας με DataFrames (παρόμοιας λογικής με SQL).

1. 1ο query σε DataFrame/SQL API

Για αυτό το query θα χρειαστεί επίσης το dataset με τα Police Stations, το οποίο και διαβάζουμε κατάλληλα. Αρχικά, από το βασικό dataset αφαιρούμε τα records που αναφέρονται στο Null Island ($LAT = 0$, $LON = 0$).Φιλτράρουμε από το βασικό dataset εκείνα τα crimes στα οποία έγινε χρήση πυροβόλου όπλου (κωδικοί που αρχίζουν από 1) και στη συνέχεια κάνουμε join με τα Police Stations ώστε κάθε crime να συνδεθεί με το αστυνομικό τμήμα που το ανέλαβε. Για να γίνει αυτό, θα χρειαστεί join με συνθήκη ισότητας στα πεδία AREA, PREC. Παρατηρούμε ότι για τους κωδικούς τοποθεσίας που είναι μικρότεροι από 10, το πεδίο AREA είναι διψήφιο (αποτελείται από 2 characters) ενώ το PREC μονοψήφιο (1 character). Για αυτό το λόγο, η συνθήκη σπάει στα δύο, μια υποσυνθήκη ισότητας των strings AREA και PREC που θα δουλέψει για διψήφιους κωδικούς μεγαλύτερους του 9 και μια υποσυνθήκη της μορφής AREA="0"+PREC (σύγκριση strings) που θα δουλέψει για κωδικούς μικρότερους του 10. Στη συνέχεια προστίθεται μια column για την απόσταση του αντίστοιχου αστυνομικού τμήματος από την τοποθεσία του crime (γίνεται χρήση της UDF συνάρτησης get_distance που φαίνεται παρακάτω η οποία χρησιμοποιεί το πακέτο geopy το οποίο αφενός γίνεται import, αφετέρου προστίθεται ως .zip dependency στα ορίσματα –py-files του pyspark). Ομαδοποιούμε ως προς year (column η οποία έχει προστεθεί ως γνωστόν), αθροίζουμε για το πλήθος των εγκλημάτων και υπολογίζουμε το μέσο distance. Η UDF συνάρτηση που αναφέρθηκε είναι η παρακάτω :

```
# calculate the distance between two points [lat1,long1], [lat2,long2] in km
def get_distance(lat1, long1, lat2, long2):
    return distance.geodesic((lat1,long1),(lat2,long2)).km

get_distance_udf=udf(get_distance,DoubleType())
```

Τα βασικά σημεία του κώδικα που επεξηγήθηκαν φαίνονται παρακάτω:

```
#keeping the crimes relevant to the desired weapon type
full_dataset_WeaponFiltered_df=full_dataset_df.filter(substring(col("Weapon Used Cd"),1,1)=="1")

#main body of the query
full_dataset_WeaponFiltered_df.join(LA_Police_Stations_df,(full_dataset_WeaponFiltered_df["AREA "]==LA_Police_Stations_df["PREC"])\
|(full_dataset_WeaponFiltered_df["AREA "]=="0"+LA_Police_Stations_df["PREC"]) )\
.withColumn("distance",get_distance_udf(col("LAT"),col("LON"),col("Y"),col("X")))\
.groupBy("year").agg(count("*").alias("#"),avg("distance").alias("average_distance")).orderBy("year").select("year","average_distance","#").show()
```

Το παραπάνω υλοποιείται και σε SQL με εντελώς παρόμοιας λογικής κώδικα, όπως φαίνεται στο github. Τα αποτελέσματα του query είναι τα εξής, στην μορφή που αναμενόταν :

```
+----+------------------+-----+
|year|  average_distance|    #|
+----+------------------+-----+
|2010|2.7836960472690264| 8161|
|2011|2.7904269556134866| 7225|
|2012| 2.834365275596899| 6521|
|2013|2.8302771057584173| 5851|
|2014|2.7128403363667344| 4257|
|2015|2.7062554193223836| 6729|
|2016| 2.717656509409668| 8094|
|2017|2.7213858844653522| 7780|
|2018|2.7355885132519306| 7414|
|2019|2.7408268365459456| 7135|
|2020|2.6880415621139844| 8492|
|2021| 2.696338747034669|12659|
|2022|2.6120866796786975|10067|
|2023|2.5561135122400156| 8796|
+----+------------------+-----+
```

Μετρώντας τους χρόνους εκτέλεσης, λαμβάνουμε ομοίως τις εξής μετρήσεις :

Μέτρηση 1	29.255775928497314
Μέτρηση 2	24.19491958618164
Μέτρηση 3	22.003702402114868
Μέτρηση 4	22.548243045806885

για τις οποίες ο μέσος όρος υπολογίζεται ως :

$$\frac{29.25 + 24.1949 + 22.003 + 22.5482}{4} = 24,5sec$$

2. 2ο query σε DataFrame/SQL API

Για αυτό το query, η λογική είναι πολύ παρόμοια με τη διαφορά ότι το grouping γίνεται ως προς αστυνομικό τμήμα ("DIVISION") και όχι ως προς year. Ομοίως αθροίζουμε το πλήθος των εγκλημάτων και υπολογίζουμε average των distances που υπολογίστηκαν με την προηγούμενη UDF.
Τα βασικά σημεία του κώδικα φαίνονται παρακάτω:

```
1 # keep crimes with not null weapon type
2 full_dataset_WeaponFiltered_df=full_dataset_df.filter(col("Weapon Used Cd"
     ).isNotNull())
3
4 #main body of the query
5 full_dataset_WeaponFiltered_df.join(LA_Police_Stations_df,(
     full_dataset_WeaponFiltered_df["AREA "]==LA_Police_Stations_df["PREC"])
      |\
6                                       (full_dataset_WeaponFiltered_df["AREA "
                                           ]=="0"+LA_Police_Stations_df["PREC"
                                           ]))\
```

```
7 .withColumn("distance",get_distance_udf(col("LAT"),col("LON"),col("Y"),col
    ("X")))\
8 .groupBy("DIVISION").agg(count("*").alias("#"),avg("distance").alias("
    average_distance")).orderBy(col("#").desc()).select("DIVISION","
    average_distance","#").show()
```

Το παραπάνω υλοποιείται και σε SQL με εντελώς παρόμοιας λογικής κώδικα, όπως φαίνεται στο github. Τα αποτελέσματα του query είναι τα εξής, στην μορφή που αναμενόταν :

```
+---------------+------------------+-----+
|       DIVISION|  average_distance|    #|
+---------------+------------------+-----+
|    77TH STREET|2.6461407076334904|94482|
|      SOUTHEAST|2.0923833705162598|77723|
|      SOUTHWEST|2.6102603087822263|72508|
|        CENTRAL| 1.007613168722048|63216|
|         NEWTON|2.0527210855337747|61156|
|        RAMPART|1.5314601017773435|55590|
|        OLYMPIC|1.7549025585082079|52773|
|      HOLLYWOOD|1.4341887142721461|50929|
|        PACIFIC| 3.878381627197251|44253|
|        MISSION| 4.704084825880415|43480|
|NORTH HOLLYWOOD|2.5449749726719264|42440|
|     HOLLENBECK|2.5926681737347312|41368|
|         HARBOR| 3.933919136132198|40658|
|       WILSHIRE| 2.401770856336865|37719|
|      NORTHEAST| 3.993418046811276|37137|
|       VAN NUYS| 2.134472548228313|36075|
|        TOPANGA|3.5113204653786556|34648|
|       FOOTHILL|4.2358687169863405|34642|
|    WEST VALLEY| 3.386252364628831|33750|
|     DEVONSHIRE|3.9810181742408366|30842|
+---------------+------------------+-----+
```

Μετρώντας τους χρόνους εκτέλεσης, λαμβάνουμε ομοίως τις εξής μετρήσεις :

Μέτρηση 1	67.49545955657959
Μέτρηση 2	65.56803250312805
Μέτρηση 3	66.50626230239868
Μέτρηση 4	69.35267329216003

για τις οποίες ο μέσος όρος υπολογίζεται ως :

$$\frac{67.495 + 65.568 + 66.506 + 69.3526}{4} = 67,23sec$$

3. 3ο query σε DataFrame/SQL API

Για αυτό το query, η λογική δεν διαφέρει πολύ αλλά τώρα κάθε έγκλημα πρέπει να συσχετιστεί με το πλησιέστερο αστυνομικό τμήμα και όχι με αυτό που το ανέλαβε. Για αυτό το λόγο, κάνουμε join του βασικού dataset με αυτό των police stations συνδέοντας κάθε έγκλημα με όλα τα αστυνομικά τμήματα και ομαδοποιούμε ανά έγκλημα υπολογίζοντας το minimum distance (μέσω aggregation), το οποίο θα αποτελεί την απόσταση μεταξύ της τοποθεσίας του crime και

του πλησιέστερου αστυνομικού τμήμα. Επειδή θέλουμε και το year, φροντίζουμε να το κρατήσουμε με ένα trivial min aggregation (καθώς όλες οι rows που ομαδοποιούνται αναφέρονται στο ίδιο crime που έγινε μια συγκεκριμένη χρονιά, οπότε όλες οι rows που ομαδοποιούνται έχουν το ίδιο year). Στη συνέχεια, ομαδοποιούμε ανά χρονιά, αθροίζουμε το πλήθος των εγκλημάτων υπολογίζοντας το average των προηγουμένως υπολογισθέντων minimum distances. Γίνεται ομοίως χρήση της γνωστής UDF για υπολογισμό απόστασης. **Σημειώνουμε** ότι για περισσότερη ευκολία, η υλοποίηση αυτού του query θα μπορούσε να γίνει με χρήση window function, ακριβώς όπως επιλέχθηκε να γίνει στο 4ο query παρακάτω (όπου θέλαμε να κρατήσουμε το πεδίο DIVISION).
Τα βασικά σημεία του κώδικα φαίνονται παρακάτω:

```
#keeping the crimes relevant to the desired weapon type
full_dataset_WeaponFiltered_df=full_dataset_df.filter(substring(col("
    Weapon Used Cd"),1,1)=="1")
#main body of the query - explained in the report
full_dataset_WeaponFiltered_with_minDistance_df=
    full_dataset_WeaponFiltered_df.join(LA_Police_Stations_df)\
.withColumn("distance",get_distance_udf(col("LAT"),col("LON"),col("Y"),col
    ("X")))\
.groupBy(col("DR_NO")).agg(min("distance").alias("minimum_distance"),min("
    year").alias("year"))

Results_df=full_dataset_WeaponFiltered_with_minDistance_df\
.groupBy("year").agg(count("*").alias("#"),avg("minimum_distance").alias("
    average_distance")).orderBy("year").select("year","average_distance","#
    ").show()
```

Το παραπάνω υλοποιείται και σε SQL με εντελώς παρόμοιας λογικής κώδικα, όπως φαίνεται στο github. Τα αποτελέσματα του query είναι τα εξής, στην μορφή που αναμενόταν :

```
+----+------------------+-----+
|year|  average_distance|    #|
+----+------------------+-----+
|2010| 2.434947736950946| 8161|
|2011|2.4582677442240515| 7225|
|2012| 2.504654649517621| 6521|
|2013| 2.459283926814491| 5851|
|2014|2.3251177284384297| 4257|
|2015| 2.388344254793133| 6729|
|2016| 2.425851163095082| 8094|
|2017|2.3903033408449064| 7780|
|2018|2.4114913328040943| 7414|
|2019| 2.430362463067704| 7135|
|2020|2.3818645038792337| 8492|
|2021|2.3516443226543347| 9746|
|2022| 2.314047537679763|10031|
|2023| 2.271752228194023| 8794|
+----+------------------+-----+
```

Μετρώντας τους χρόνους εκτέλεσης, λαμβάνουμε ομοίως τις εξής μετρήσεις :

Μέτρηση 1	135.8511769771576
Μέτρηση 2	133.88080191612244

για τις οποίες ο μέσος όρος υπολογίζεται ως :

$$\frac{135.85 + 133.88}{2} = 134,866 sec$$

4. 4ο query σε DataFrame/SQL API

Για αυτό το query, η λογική είναι παρόμοια με το προηγούμενο query αλλά η ομαδοποίηση θα πρέπει να γίνει ανά division και όχι ανά year. Ωστόσο τα πράγματα δεν είναι τόσο προφανή εδώ. Αν χρησιμοποιήσουμε το σώμα του προηγούμενου query, το πλεονέκτημα ότι μετά το join τα records ενός crime είχαν το ιδιο year και με ένα trivial min μπορούσαμε να το κρατήσουμε εδώ δεν υπάρχει. Θέλουμε με κάποιον τρόπο να κρατήσουμε το division του police station για το οποίο συναντάται το min distance. Αυτό είτε πρέπει να υλοποιηθεί με ένα επιπρόσθετο join με συνθήκη ισότητας το distance να είναι το υπολογισθέν minimum distance (υλοποίηση που φαίνεται σε σχόλιο παρακάτω), είτε πρέπει να χρησιμοποιηθεί window fucntion όπως και επιλέγεται για λόγους κομψότητας του κώδικα (μιας και η επίδοση δεν διαφοροποιήθηκε ιδιαίτερα όπως παρατηρήσαμε). Αφού φιλτράρουμε τα crimes με μη κενό wepaon type, κάνουμε join με τα police stations υλοποιίζοντας το distance τους. Κάθε crime τώρα συνδέεται με όλα τα police stations και με την απόσταση από αυτά. Με χρήση της row_number() και του ορισθέντος window specification, κάθε row για ένα συγκεκριμένο crime (partition by DR_NO) αποκτά ένα ranking που αφορά το πόσο κοντινό είναι ένα αστυνομικό τμήμα από αυτό. Για κάθε crime, το row που το διασυνδέει με το κοντινότερο αστυνομικό τμήμα αποκτά ranking 1, το row που το διασυνδέει με το δεύτερο κοντινότερο αστυνομικό τμήμα αποκτά ranking 2 κ.ο.κ. Αυτό δε θα μπορούσε να γίνει με group by κα aggregations στα οποία ένα σύνολο rows αντιστοιχίζεται σε ένα αθροιστικό row. Εδώ κρατάμε όλα τα rows με μια παραπάνω πληροφορία, το rank. Φιλτράρουμε τα rows με rank=1 (άρα κρατάμε τις καταχωρήσεις που αφορούν τα πλησιέστερα αστυνομικά τμήματα) και στη συνέχεια ομαδοποιούμε ως προς DIVISION, αθροίζουμε το πλήθος των crimes και υπολιγίζουμε average των distances.
Τα βασικά σημεία του κώδικα φαίνονται παρακάτω:

```
#keep crimes with not null weapon type
full_dataset_WeaponFiltered_df=full_dataset_df.filter(col("Weapon Used Cd").isNotNull())

full_dataset_WeaponFiltered_with_distances_df=full_dataset_WeaponFiltered_df.join(LA_Police_Stations_df)\
.withColumn("distance",get_distance_udf(col("LAT"),col("LON"),col("Y"),col("X")))

#defining the window specification
window_spec = Window.partitionBy("DR_NO").orderBy("distance")

full_dataset_WeaponFiltered_with_distances_df.withColumn("rank",row_number().over(window_spec)).filter(col("rank")==1)\
.groupBy("DIVISION").agg(count("*").alias("#"),avg("distance").alias("average_distance"))\
.orderBy(col("#").desc()).select("DIVISION","average_distance","#").show()

end=time.time()
print("Execution Time:",end-start)

```

```
spark.stop()

'''
#OTHER IMPLEMENTATION : USING ONE MORE JOIN IN ORDER TO KEEP THE DIVISION FIELD
full_dataset_WeaponFiltered_with_minDistance_df= full_dataset_WeaponFiltered_with_distances_df\
.groupBy(col("DR_NO")).agg(min("distance").alias("minimum_distance")).alias("first")\
.join(full_dataset_WeaponFiltered_with_distances_df.alias("second"),(col("first.DR_NO")==col("second.DR_NO")) &\
    (col("first.minimum_distance")==col("second.distance")))\
.select(col("first.DR_NO"), col("second.DIVISION"),col("first.minimum_distance"))


Results_df=full_dataset_WeaponFiltered_with_minDistance_df\
.groupBy("DIVISION").agg(count("*").alias("#"),avg("minimum_distance").alias("average_distance")).orderBy(col("#").desc()).select("DIVISION","average_distance","#").show()
'''
```

Το παραπάνω υλοποιείται και σε SQL με εντελώς παρόμοιας λογικής κώδικα, όπως φαίνεται στο github. Τα αποτελέσματα του query είναι τα εξής, στην μορφή που αναμενόταν :

```
+----------------+------------------+-----+
|        DIVISION|  average_distance|    #|
+----------------+------------------+-----+
|     77TH STREET|1.6723675400753428|78788|
|       SOUTHWEST|2.1610204753228954|78052|
|       HOLLYWOOD|  1.92045334592717|70630|
|       SOUTHEAST|2.2230876117786127|66696|
|         OLYMPIC| 1.664963565226659|60785|
|         CENTRAL|0.8642106412913209|59191|
|        WILSHIRE|2.4786980950047433|58093|
|         RAMPART| 1.359765638381019|56307|
|        VAN NUYS|2.8057293845619724|55261|
|          NEWTON|1.5991783687690007|45397|
|      HOLLENBECK|2.5846615655581577|42032|
|         PACIFIC| 3.845963304265378|40360|
| NORTH HOLLYWOOD|2.5933785029357086|40198|
|          HARBOR| 3.680883514111903|39433|
|        FOOTHILL| 3.978177413359706|38246|
|     WEST VALLEY| 2.832624129255953|34570|
|         TOPANGA|3.0580394309881895|32867|
|       NORTHEAST| 3.769628808555916|27270|
|         MISSION| 3.777109475878288|27086|
|WEST LOS ANGELES| 2.713325375278826|22373|
+----------------+------------------+-----+
```

Μετρώντας τους χρόνους εκτέλεσης, λαμβάνουμε ομοίως τις εξής μετρήσεις :

Μέτρηση 1	1206.1014604568481
Μέτρηση 2	1303.6401064395905

για τις οποίες ο μέσος όρος υπολογίζεται ως :

$$\frac{1206.1014 + 1303.64}{2} = 1254,87sec$$

5. Συμπεράσματα

Στη συνέχεια, επιχειρούμε να συμπεράνουμε κατά πόσο τα διάφορα εγκλήματα τα ανέλαβαν τα κοντινότερα σε αυτά αστυνομικά τμήματα (σύγκριση αποτελεσμάτων των queries 2 και 4), καθώς και με ποιον τρόπο αυτή η συμπεριφορά (το να αναλαμβάνουν τα αστυνομικά τμήματα ή όχι τα κοντινότερά τους εγκλήματα) επαναλαμβάνεται ή όχι με τον ίδιο τρόπο ανά τα χρόνια (σύγκριση αποτελεσμάτων των queries 1 και 3).
Από τη σύγκριση των αποτελεσμάτων των queries 2 και 4 συμπεραίνουμε τα εξής χρησιμοποιώντας δύο χαρακτηριστικά παραδείγματα :

1. Το αστυνομικό τμήμα "77 TH STREET" ανέλαβε 94482 crimes (από query 2) ενώ τα κοντινότερα σε αυτό crimes ήταν λιγότερα, 78788 (από query 4). Συνεπώς, από αυτο μπορούμε να συμπεράνουμε με ασφάλεια ότι το συγκεκριμένο αστυνομικό τμήμα ανέλαβε τουλάχιστον 15694 crimes τα οποία δεν είναι πλησιέστερα σε αυτό. Σίγουρα δηλαδή ανέλαβε crimes τα οποία δεν είναι κοντινότερά του χωρίς ταυτόχρονα να μπορούμε να εγγυηθούμε ότι τουλάχιστον ανέλαβε όλα τα κοντινότερά του και μετά αποφάσισε να αναλάβει και κάποια πιο μακρινά. Αυτή η συμπεριφορά συναντάνται και σε άλλα αστυνομικά τμήματα.

2. Το αστυνομικό τμήμα "HOLLYWOOD" ανέλαβε 50929 crimes (από query 2) ενώ τα κοντινότερα σε αυτό crimes ήταν περισσότερα, 70630 (από query 4). Ως εκ τούτου, το συγκεκριμένο αστυνομικό τμήμα δεν ανέλαβε όλα τα πλησιέστερα σε αυτό crimes, επομένως υπήρξαν crimes (τουλάχιστον $70630 - 50929 = 19701$) που δεν τα ανέλαβε το κοντινότερο αστυνομικό τμήμα. Αυτή η συμπεριφορά συναντάται και σε άλλα αστυνομικά τμήματα.

Συνεπώς από τα παραπάνω, συμπεραίνουμε ότι τα αστυνομικά τμήματα έχουν την τάση να αναλαμβάνουν εγκλήματα που δεν είναι πλησιέστερα σε αυτά και αντιστρόφως υπάρχουν εγκλήματα (αρκετά) τα οποία δεν τα ανέλαβε το πλησιέστερο αστυνομικό τμήμα.
Εξετάζουμε στη συνέχεια αν αυτή η συμπεριφορά επαναλαμβάνεται με παρόμοιο τρόπο ανά τα χρόνια (από 2010 έως 2023). Από τη σύγκριση των αποτελεσμάτων των queries 1 και 3 παρατηρούμε σε κάθε χρονιά η μέση απόσταση των crime locations από τα αστυνομικά τμήματα που τα ανέλαβαν (από query 1) είναι μεγαλύτερη από την μέση απόστασή τους από τα πλησιέστερα αστυνομικά τμήματα (από query 3), κάτι που σημαίνει ότι κάθε χρονιά υπάρχει η τάση αυτή η συμπεριφορά που περιεγράφηκε (κατά την οποία εγκλήματα αναλαμβάνονται από μη πλησιέστερα αστυνομικά τμήματα) να επαναλαμβάνεται. Μάλιστα για κάθε χρονιά, αυτή η απόκλιση είναι παρόμοια. Συνεπώς, σε κάθε χρονιά αυτή η συμπεριφορά επαναλαμβάνεται με παρόμοιο τρόπο.

VII. Ερώτημα 7^o : Σύγκριση Μεθόδων Υλοποίησης των Joins από το Spark

Στο ερώτημα αυτό, εξετάζουμε τις διαφορετικές μεθόδους υλοποίησης των Joins από το Spark. Αφού παρουσιάσουμε την λειτουργία και το οφέλη χρήσης της καθεμιάς από τις μεθόδους, παρατηρούμε τον τρόπο με τον οποίο επηρεάζουν την επίδοση στην εκτέλεση των queries 3 και 4. Παρουσιάζουμε κατάλληλα visualized σχήματα από τον History Server και τμήματα από query execution plans που παράχθηκαν μέσω της explain.

Hash Join Οι πρώτες δύο από τις επόμενες μεθόδους χρησιμοποιούν Hash Join, στο οποίο δημιουργείται ένα hash table (με βάση το join key) στον μικρότερο από τους δύο πίνακες του join (έτσι, η προσπέλαση των records του γίνεται σε O(1)) και για την εύρεση records που ταιριάζουν στο join key, λουπάρουμε στα records του μεγάλου πίνακα, ελέγχοντας ταυτόχρονα για matches μέσω του hash table του μικρού πίνακα.

1. Broadcast Hash Join : BHJ

Με τη μέθοδο αυτή, ο ένας από τους δύο πίνακες του join (όπως θα δούμε, ο μικρότερος) στέλνεται με broadcast messages σε όλους τους κόμβους (executors) και ο δέυτερος πίνακας γίνεται partitioned και στέλνεται κατά τμήματα σε διαφορετικούς κόμβους (πιθανώς σε ένα κατανεμημένο περιβάλλον, η κατανομή αυτού του πίνακα θα έχει ήδη γίνει). Για λόγους εξοικονόμησης χώρου και network bandwidth (μιας και ο broadcasted πίνακας στέλνεται μέσω δικτύου σε όλους), προφανώς ο μικρός πίνακας είναι αυτός που στέλνεται ως αντίγραφο σε όλους και ο μεγάλος διασπάται. Στη συνέχεια, ο κάθε κόμβος, για τα τμήματα των πινάκων που περιέχει, εκτελεί hash join, όπως αυτό περιεγράφηκε [Εικόνα 10]. Από τον τρόπο λει-

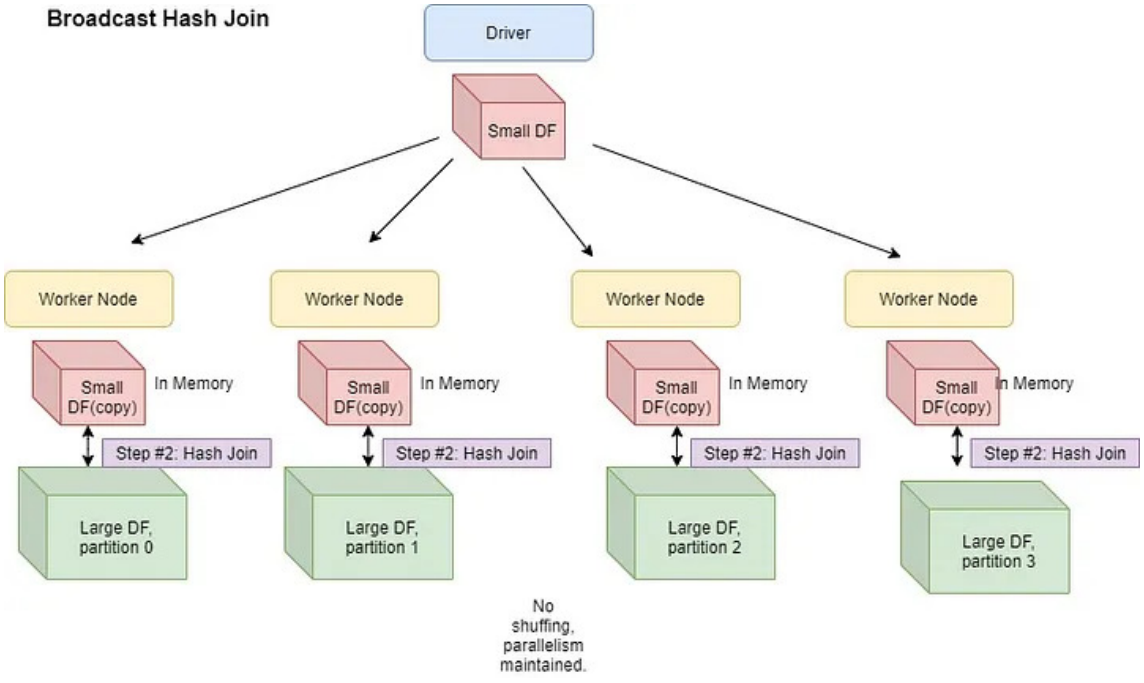

Figure 10: Broadcast Hash Join

τουργίας του, γίνεται εμφανές ότι χρησιμοποιείται όταν ο ένας εκ των δύο πινάκων είναι αρκετά μικρός ώστε να γίνεται broadcast χωρίς μεγάλο δικτυακό κόστος (και πρόκληση Out Of Memory errors στους κόμβους) και όταν το network bandwidth είναι επαρκές. Επιπλέον, το πλήθος των executors μπορεί να επηρεάσει την επίδοση αφού όσο μεγαλύτερο είναι, τόσο πιο ακριβό θα είναι το broadcast σε όλους αυτούς. Τέλος, προφανώς θα θέλαμε ο ένας εκ των δύο πινάκων που είναι partitioned, να είναι ήδη εξαρχής ισοκατανεμημένος μεταξύ των κόμβων (για load balancing). Στο Spark, εφαρμόζοντας αυτόν τον τύπο Join με χρήση της hint (για παράδειγμα στο πρώτο join του query 3), βλέπουμε στο παρακάτω τμήμα του query execution plan (εικόνα 11) ότι η δεξιά σχέση που αντιστοιχεί στον μικρό πίνακα, γίνεται broadcast σε 3 executors (3 εξερχόμενες ακμές από το "BroadcastExchange"). Ο ένας executor ηδη διαθέτει τον μικρό πίνακα και τον μοιράζει στους άλλους 3 (χρησιμοποιούμε 4 executors για την εκτέλεση). Στο τέλος μιας εκ των τριών ακμών βλέπουμε ότι εντός ενός εκ των executors γίνεται HashJoin όπως περιεγράφηκε. Για λόγους πληρότητας, επισημαίνουμε ότι η αριστερή σχέση που προκύπτει από union είναι το βασικό μας dataset.

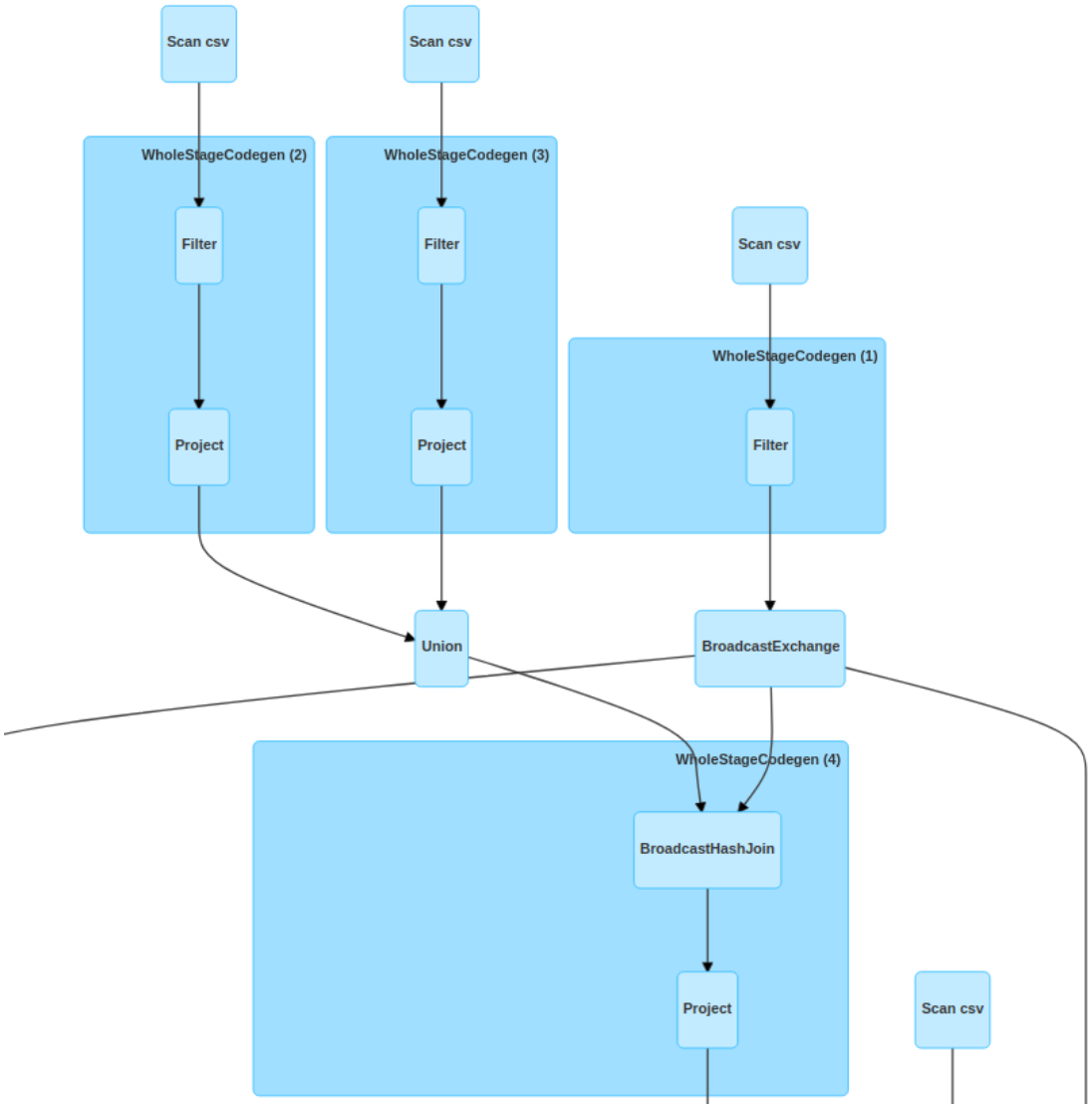

Figure 11: Broadcast Hash Join - Query Execution Plan

2. Shuffle Hash Join : SHJ

Η μέθοδος αυτή ξεκινά με τη διαδικασία του shuffling κατά το οποίο και οι δύο πίνακες κατανέμονται κατάλληλα μεταξύ των executors. Records και των δύο πινάκων με συγκεκριμένο value του join key πηγαίνουν στον ίδιο κόμβο (executor σε μας), όπως φαίνεται και στην εικόνα 12. Στη συνέχεια, εντός κάθε κόμβου γίνεται hash join μεταξύ των τμημάτων των δύο πινάκων που έχει κάθε executor. Η διαφορά με την προηγούμενη μέθοδο είναι το shuffling έναντι του broadcast. Η παραπάνω μέθοδος ενδείκνυται σε περιπτώσεις αρκετά μεγάλων πινάκων που

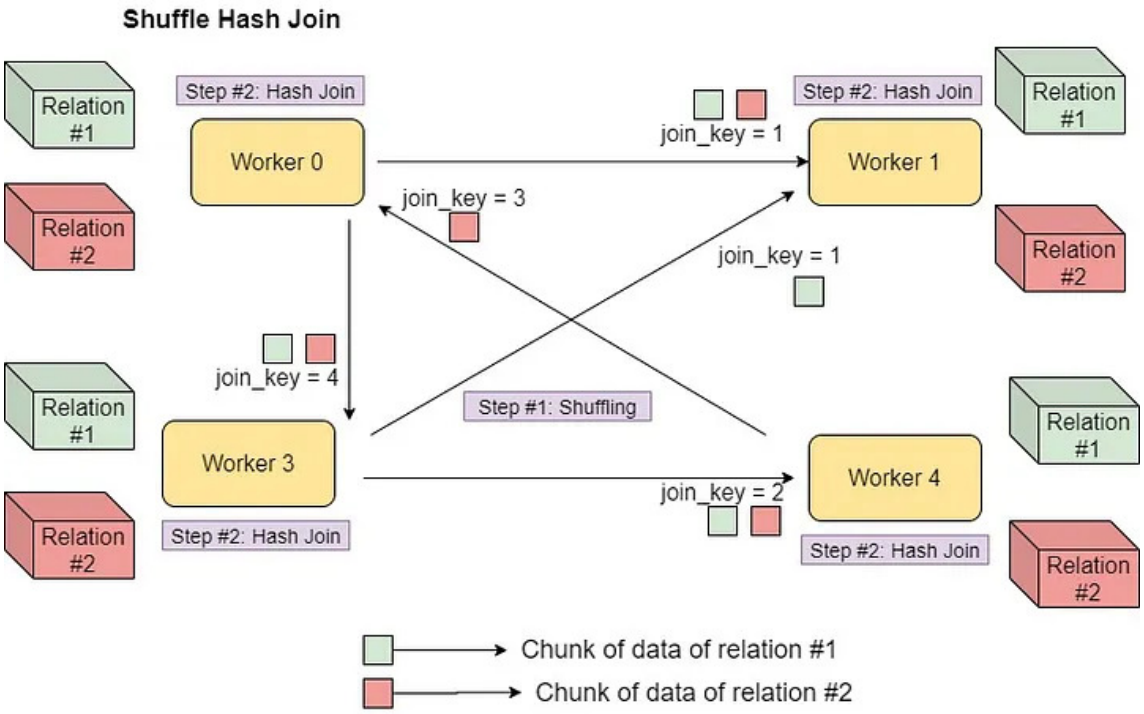

Figure 12: Shuffle Hash Join

γίνονται Join (συνήθως δεν κάνουν fit στην memory του ενός κόμβου) και δεν θα ήταν εφικτός ο broadcast διαμοιρασμός τους. Επιπλέον, δεδομένου του τρόπου με τον οποίο γίνεται το shuffling, χρησιμοποιείται σε περιπτώσεις που τα records είναι έτσι δομημένα ώστε να μπορούν να διαμοιραστούν ισοκατανεμημένα βάσει του join key (δηλαδή join keys not skewed). Κάτι που επιπλέον θα βοηθούσε είναι να έχει ήδη γίνει από πριν ένα καλό data distribution που θα διευκόλυνε το shuffling. Χρησιμοποιώντας το παραπάνω join στο Spark, παίρνουμε την εικόνα 13 από το execution plan. Όπως φαίνεται, καθένας από τους δύο πίνακες γίνεται shuffle (AQEShuffleRead) και στη συνέχεια γίνεται το hash join ("ShuffleHashJoin").

3. Shuffle Sort Merge Join : SMJ

Με τη μέθοδο αυτή, αφού πρώτα γίνει η διαδικασία του shuffling όπως εξηγήθηκε, στη συνέχεια ο κάθε κόμβος εσωτερικά δεν εκτελεί hash join αλλά sort merge join. Τα τμήματα των δύο πινάκων που ο εκάστοτε κόμβος έχει συγκεντρώσει, ταξινομούνται βάσει του join key και στη συνέχεια γίνεται το matching των records μέσω του merge, το οποίο μοιάζει με αυτό του αλγορίθμου για mergesort [Εικόνα 14] (και ως εκ τούτου έχει πολυπλοκότητα ανάλογη του μεγέθους των υποπινάκων). Συνολικά η μέθοδος δουλεύει όπως στην εικόνα 15. Η μέθοδος

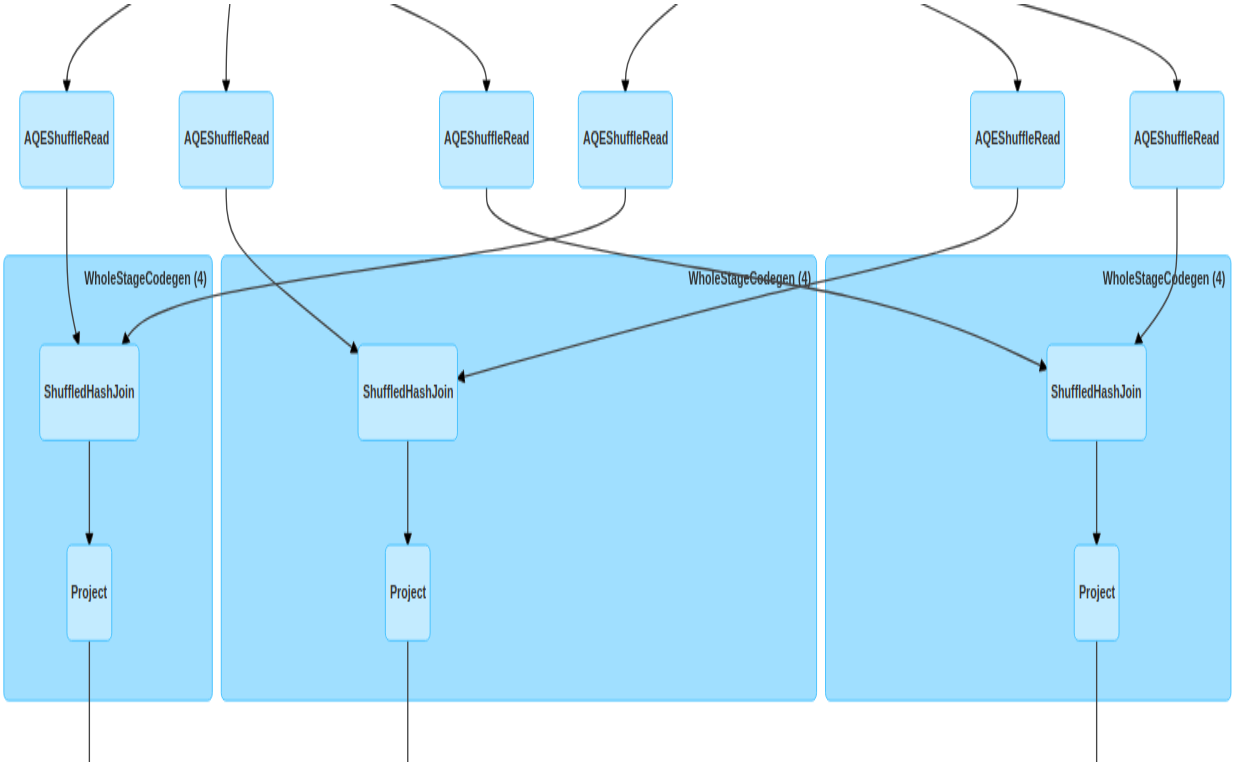

Figure 13: Shuffle Hash Join - Query Execution Plan

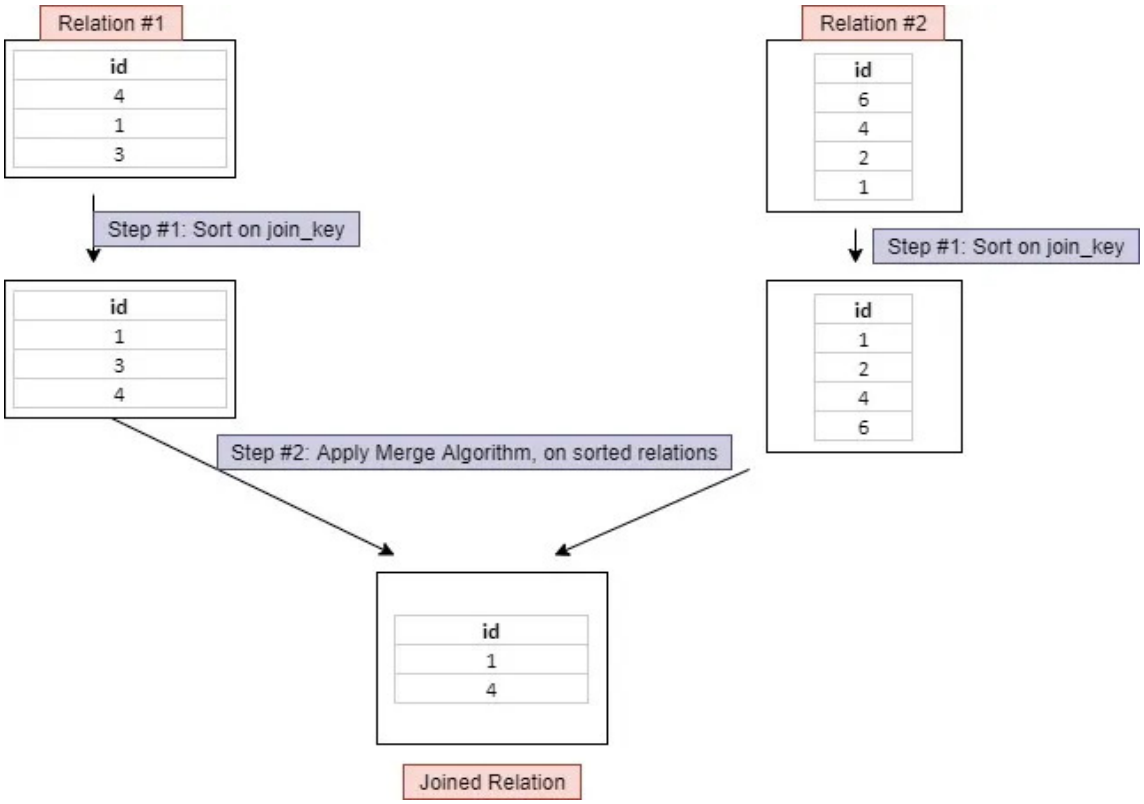

Figure 14: Sort Merge Join

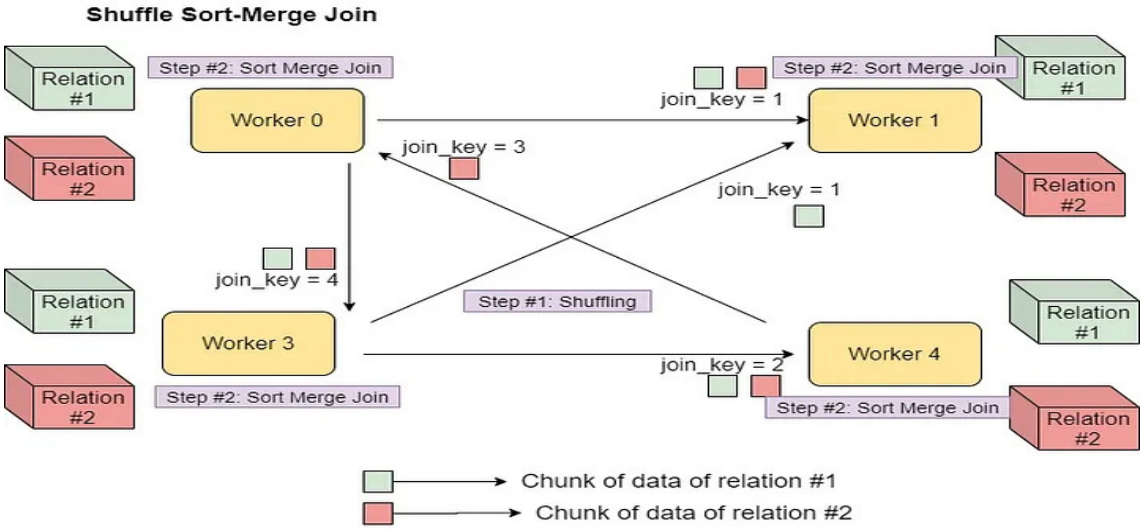

Figure 15: Shuffle Sort Merge Join

αυτή επιλέγεται αφενός όταν οι πίνακες δεν είναι μικροί (και έτσι δεν μπορεί να εφαρμοστεί BHJ) αλλά ταυτόχρονα δεν είναι και υπερβολικά μεγάλοι, τόσο ώστε η διαδικασία του sorting να γίνει ακριβή. Χρησιμοποιείται εν ολίγοις για ενδιάμεσου μεγέθους πίνακες. Η SMJ μπορεί να είναι καλύτερη από την SHJ όταν οι πίνακες δεν είναι πολύ μεγάλου μεγέθους, οπότε και το sorting είναι λιγότερο ακριβό από την δημιουργία του hash table και το looping επί του μεγάλου υποπίνακα (δηλαδή από το hash join). Χρησιμοποιώντας αυτή τη μέθοδο στο Spark, βλέπουμε το τμήμα του execution plan στην εικόνα 16. Αφού γίνει το shuffling ("AQEShuffleRead"), στη συνέχεια κάθε executor κάνει "Sort" και τους δύο υποπίνακες που διαθέτει και στη συνέχεια "SortMergeJoin" όπως φαίνεται στο διάγραμμα.

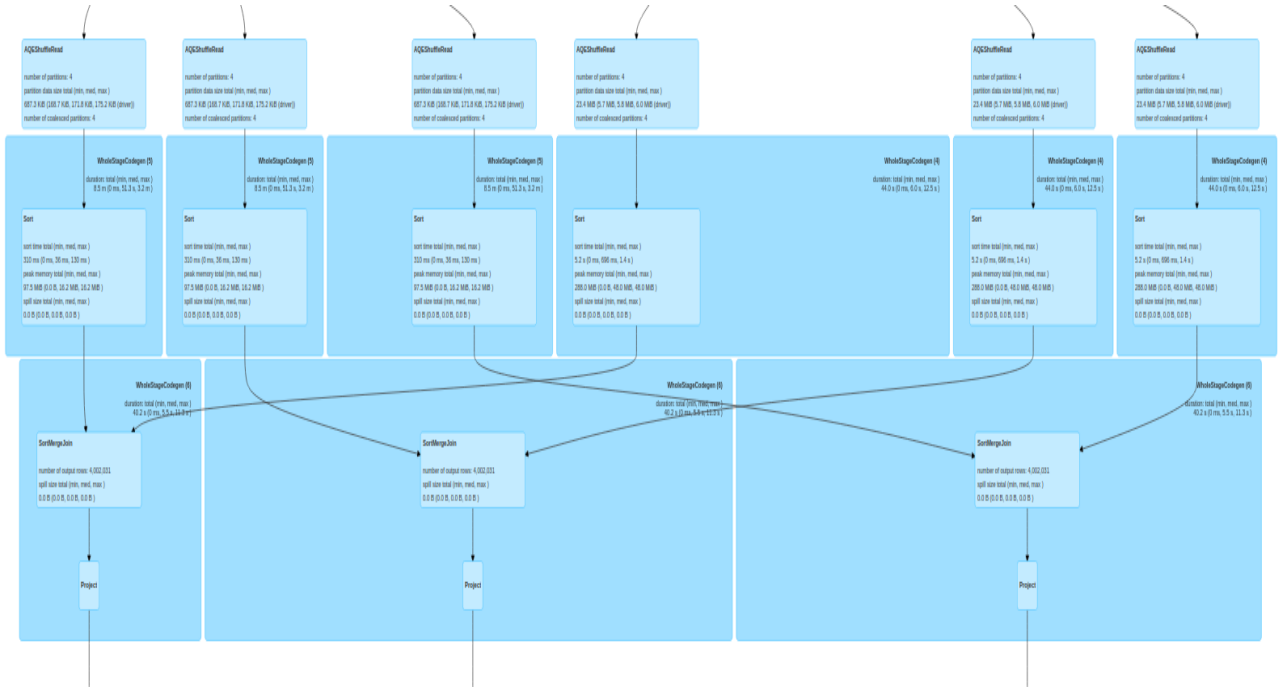

Figure 16: Shuffle Sort Merge Join - Query Execution Plan

4. Shuffle and Replication Nested Loop Join (Cartesian Product Join) : CPJ

Αυτή η μέθοδος ακολουθεί λίγο διαφορετική προσέγγιση και δεν περιλαμβάνει shuffling ούτε broadcast (στη δεύτερη περίπτωση θα ήταν broadcast nested loop join, μέθοδος που δεν μελετάται). Οι δύο πίνακες έστω A, B που συμμετέχουν στο Join, χωρίζονται σε partitions. Όλα τα partitions του ενός dataset (έστω B) στέλνοται στα partitions του άλλου dataset (A) ώστε κάθε executor που θα έχει ένα partition του ενός (A) και όλα τα partitions του άλλου dataset (B) να εκτελέσει σε αυτά nested loop join, δηλαδή εξαντλητικό cartesian join (για κάθε record του ενός dataset/partition A ελέγχεται αν γίνεται match με όλα τα records όλων των partitions του δεύτερου dataset B) [Εικόνα 17]. Χρησιμοποιώντας τη μέθοδο αυτή

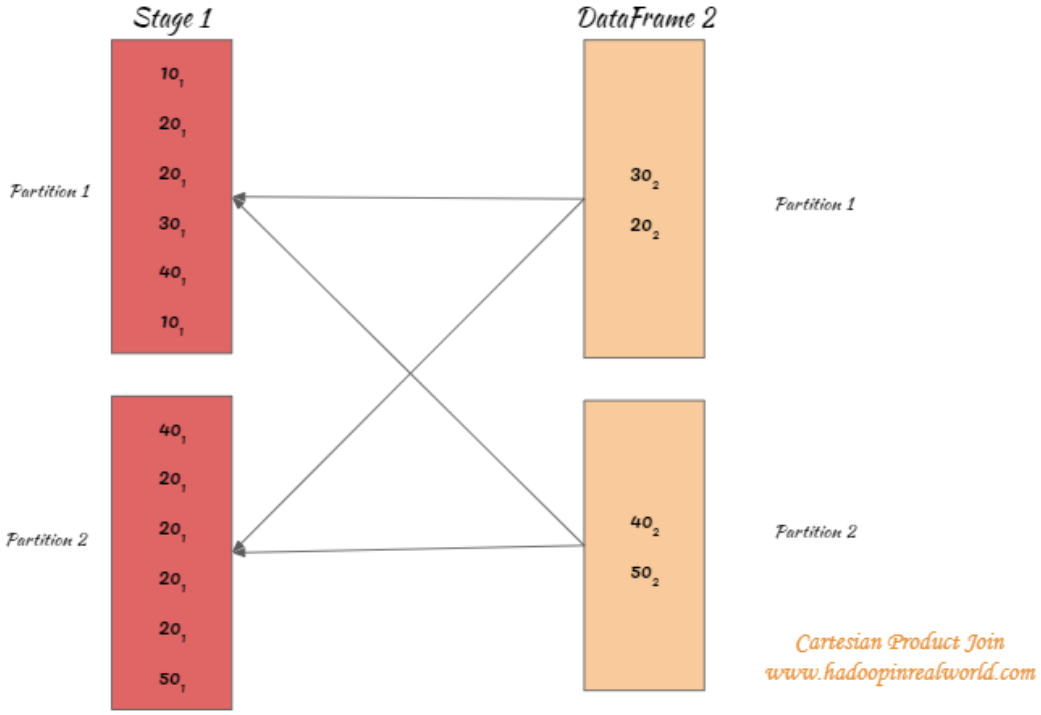

Figure 17: Cartesian Product Join

στο Spark, λαμβάνουμε το παρακάτω execution plan της εικόνας 18, όπου φαίνεται η εκτέλεση του σταδίου "CartesianProduct". Η μέθοδος αυτή θεωρητικά (και πρακτικά όπως θα δούμε) είναι η ακριβότερη όλων καθώς δημιουργεί όλες τους δυνατούς συνδυασμούς από records (π.χ. όταν εκτελεί το nested loop join). Δεν είναι τυχαίο που στην default λειτουργία, το spark τοποθετεί αυτή την μέθοδο ως την χείριστη προς επιλογή, τελευταία σε προτεραιότητα.

5. Χρησιμοποίηση των Μεθόδων στα Queries 3 και 4 και Επιδόσεις

Στη συνέχεια, χρησιμοποιούμε τις παραπάνω μεθόδους υλοποίησης των joins, στα joins των queries 3 και 4 (για το query 4 ενδεικτικά στο πρώτο από τα 4 υποερωτήματα). Εξετάζουμε τον χρόνο εκτέλεσης των queries και εμφανίζουμε το query plan με explain (ενδειτικά στο πρώτο join του query 3 που περιέχει πολλαπλά joins). Για το **Query 3**, οι χρόνοι εκτέλεσης ανά

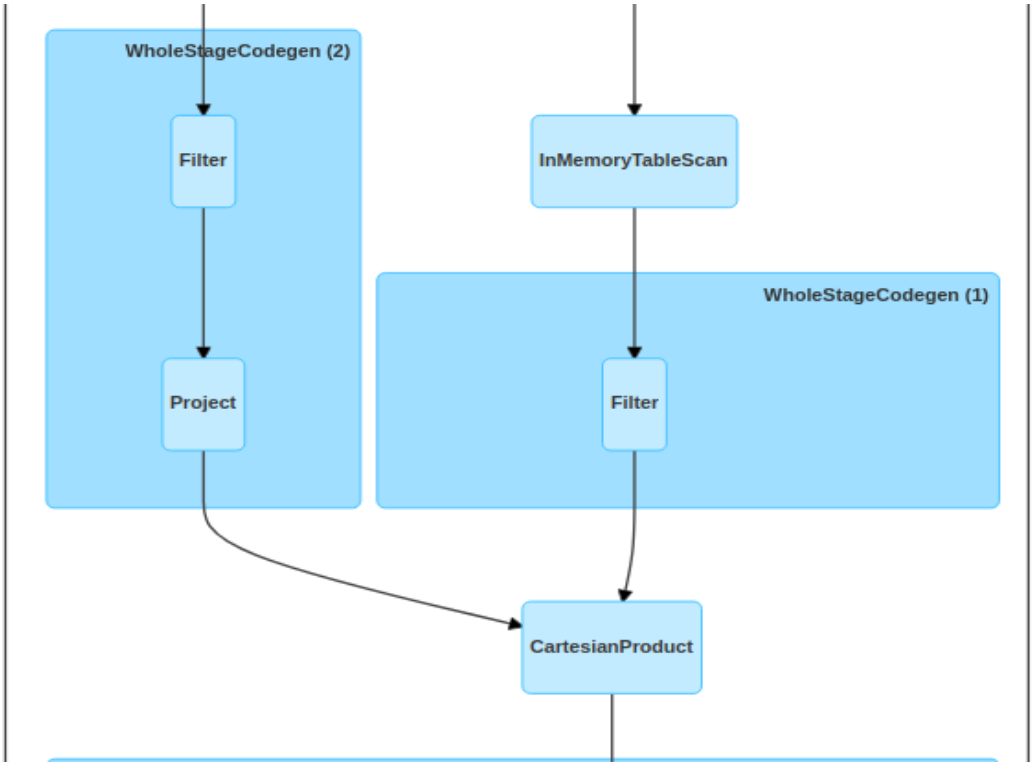

Figure 18: Cartesian Product Join - Query Execution Plan

μέθοδο join είναι οι παρακάτω. Παρατηρούμε ότι η καλύτερη μέθοδος είναι αυτή του Broadcast

Μέθοδος	**Χρόνος(sec)**
BHJ	43.7626
SHJ	48.075
SMJ	44.685375
CPJ	76.09438

Figure 19: Χρόνοι Εκτέλεσης Query 3 με χρήση διαφορετικών τύπων Spark Join

Hash Join, ακολουθούν οι SMJ (με πολύ μικρή διαφορά) και SHJ (με μεγαλύτερη διαφορά) ενώ η Shuffle and Replication NL είναι με διαφορά η χειρότερη όλων, για λόγους που εξηγήθηκαν παραπάνω. Στο συγκεκριμένο query, γίνονται 4 joins στο οποίο συμμετέχουν : το βασικό dataset (2.98 millions of rows), το revgecoding dataset (37782 rows), το LAincome2015 (285 rows) και δύο βοηθητικοί πίνακες (με τα τα ZIP codes των 3 πλυοσιότερων και 3 φτωχότερων περιοχών) οι οποίοι είναι πολύ μικροί (από 3 rows ο καθένας). Τα joins με τους δύο τελευταιούς μικρούς πίνακες ευνοούνται από το BHJ ενώ τα joins με τους revgecoding και LAincome2015 που είναι μεσαίου μεγέθους (σε σχέση με το βασικό dataset) οριακά ευνοούνται από το BHJ. Σε κάθε περίπτωση στα joins που γίνονται συμμετέχει τουλάχιστον ένας μικρός προς μεσαίο πίνακα και το BHJ έχει ικανοποιητική επίδοση. Ωστόσο, επειδή κάποιοι πίνακες είναι περισσότερο μεσαίου παρά μικρού μεγέθους, το SMJ έχει επίσης αρκετά ικανοποιητική επίδοση (καθώς γίνεται ένα μόνο δύσκολο sorting του μεγάλου βασικού dataset και στη συνέχεια sortings μικρότερων πινάκων) σε αντίθεση με το SHJ που είναι χειρότερο. Ένα βασικό μειονέκτημα των SHJ, SMJ είναι φυσικά το κόστος του shuffling που βέβαια επιβαρύνει τις δύο μεθόδους

συγκριτικά με το BHJ. Το CPJ για ευνόητους λόγους είναι το χειρότερο.
Για το **Query 4**, οι χρόνοι εκτέλεσης ανά μέθοδο join είναι οι παρακάτω. Παρατηρούμε ότι το

Μέθοδος	**Χρόνος(sec)**
BHJ	28.4917
SHJ	38.5231
SMJ	45.9781
CPJ	32.4774

Figure 20: Χρόνοι Εκτέλεσης Query 4 με χρήση διαφορετικών τύπων Spark Join

BHJ επιτυγχάνει την καλύτερη επίδοση, δεύτερο έρχεται (ως έκπληξη με την πρώτη ματιά) το CPJ και ακολουθούν τα SMJ και έπειτα το SHJ που είναι και το χειρότερο για αυτό το query. Στο query αυτό γίνεται join μεταξύ του αρχικού μεγάλου dataset (φιλτραρισμένου μεν αλλά ακόμη μεγάλου) και των LA Police Stations που αποτελούν ένα πολύ μικρό dataset (21 rows!), καθιστώντας το BHJ ως το καλύτερο δυνατό join. Το CPJ έρχεται δεύτερο καθώς λόγω του πολύ μικρού μεγέθους του δεύτερου dataset το εξαντλητικό nested loop join ουσιαστικά θα χρειαστεί να σκανάρει 21 rows (έναν μικρό σταθερό αριθμό από rows δηλαδή) για κάθε row του μεγάλου dataset, κάτι που δεν αποτελεί πολλή δουλειά αλλά βέβαια δεν καταφέρνει να εκμεταλλευτεί τα οφέλη του broadcast hash join. Το SMJ αποδεικνύεται χειρότερο από το SHJ καθώς επειδή ο δεύτερος (ο μικρότερος από τους δύο) πίνακας είναι αρκετά μικρός, η δημιουργία του hash table δεν είναι καθόλου ακριβή, σε αντίθεση με το sorting του μεγάλου dataset που θα πρέπει να εκτελέσει η SMJ. Τέλος με χρήση της explain, παρουσιάζονται ενδεικτικά τμήματα των query plans όπου φαίνεται η μέθοδος υλποίησης των joins.

BHJ BuildRight : Χρησιμοποιήθηκε η δεξιά σχέση για την δημιουργία του hash table και αυτή έγινε broadcast.

```
+- BroadcastHashJoin [knownfloatingpointnormalized(normalizenanandzero(LAT
   #5940)), knownfloatingpointnormalized(normalizenanandzero(LON#5969))],
   [knownfloatingpointnormalized(normalizenanandzero(cast(LAT#6290 as
   double))), knownfloatingpointnormalized(normalizenanandzero(cast(LON
   #6291 as double)))], LeftOuter, BuildRight, false
```

SMJ Για την Shuffle Sort Merge Join

```
+- SortMergeJoin [knownfloatingpointnormalized(normalizenanandzero(LAT
   #9764)), knownfloatingpointnormalized(normalizenanandzero(LON#9793))],
   [knownfloatingpointnormalized(normalizenanandzero(cast(LAT#10114 as
   double))), knownfloatingpointnormalized(normalizenanandzero(cast(LON
   #10115 as double)))], LeftOuter
```

SHJ Για την Shuffle Hash Join

```
+- ShuffledHashJoin [ZIPcode#16223], [Zip Code#16193], Inner, BuildRight
```

CPJ Για την Cartesian Product Join

```
+- CartesianProduct (ZIPcode#19143 = Zip Code#19113)
```

References

[1] Comparing Spark SQL and DataFrame API: Differences and Use Cases (https://www.sparkcodehub.com/spark-sql-vs-dataframe-api)

[2] RDDs vs. Dataframes vs. Datasets – What is the Difference and Why Should Data Engineers Care? (https://www.analyticsvidhya.com/blog/2020/11/what-is-the-difference-between-rdds-dataframes-and-datasets/)

[3] Spark RDD vs DataFrame vs Dataset (https://sparkbyexamples.com/spark/spark-rdd-vs-dataframe-vs-dataset/)

[4] Lecture Slides: "Introduction to Apache Spark" (slide 53) (https://helios.ntua.gr/pluginfile.php/175114/mod_resource/content/3/MainMemoryProcessing-Spark.pdf - *NTUA authentication is required*)

[5] Spark Join Strategies — How & What? (https://towardsdatascience.com/strategies-of-spark-join-c0e7b4572bcf)

[6] Different Types of Spark Join Strategies (https://medium.com/@ongchengjie/different-types-of-spark-join-strategies-997671fbf6b0)

[7] A Deep Dive into Apache Spark Join Strategies (https://python.plainenglish.io/a-deep-dive-into-the-inner-workings-of-apache-spark-join-strategies-5d47dd29a4cb)

[8] How does Cartesian Product Join work in Spark? (https://www.bigdatainrealworld.com/how-does-cartesian-product-join-work-in-spark/)